КОМПЮТЪРНИ АРХИТЕКТУРИ

Цветан Димитров Таслаков

Научна степен: кандидат на науките

Тема: Изследване и алгоритмизация на един клас паралелни методи за решаване на системи линейни алгебрични уравнения и изчислителни структури

Анотация: Разработени са паралелни методи /алгоритми/ и са обвързани с подходяща структура на паралелната изчислителна система. Получени са максималните възможни коефициенти на бързодействие и ефективност. Използуван е математическия апарат на линейната алгебра за анализ на паралелните методи и мрежи на петри за анализ на работата на паралелната система. Предложени са паралелни методи и алгоритми за решаване на системи линейни алгебрични уравнения и два алгоритъма за разпределение на заданията между процесорите; формулирани са изисквания към паралелната изчислителна система; развит е подходящ аналитичен модел за оценка на бързодействието и ефективността; разработена е паралелна изчислителна система с двойна комуникационна памет и е съкратено времето за обмен. Прил. 10, табл. 11, граф. 6, сх. 34 09

Година на защита: 1987

1. Въведение в дисциплината

2. Особености в архитектурата на съвременните компютри.

CISC и RISCпроцесори. Паралелизъм в работата на процесора. Шината като основна среда за трансфер на данни и команди.

3. Въведение в паралелната обработка.

Необходимост от паралелна обработка. Нива на паралелност. Модели на мащабируемостта. Класификация на паралелните компютри. Разпределена обработка.

4. Конвейерно изпълнение на командите в процесора.

Основни принципи на конвейерната обработка. Особености на конвейера за команди. Проблеми при конвейерното изпълнение на команди. Работа на конвейера при изпълнение на команди за преход. Междукомандни зависимости – същност на междукомандните зависимости, отстраняване на междукомандните зависимости. Пример: работа на конвейерите в процесорните архитектури на Intel P5 и P6 и процесора P4P.

5. Процесор с множество функционални устройства.

Въведение. Синхронизация на апаратно ниво. Синхронизация на програмно ниво. Пример: Процесор на Intel Itanium 2. Сравнение на двата подхода за синхронизация.

6. Векторни процесори.

Структура на векторните процесори. Векторни команди – код на операциите, адресация на операндите, особености при фиксиране на състоянието след изпълнение командите.

7. Паралелни компютри с разпределена памет.

SMP и MPP компютри. Абстрактен модел на изчисление в MPP компютрите. Проблеми на компютрите с разпределена памет.

8. Комуникационни мрежи.

Въведение. Характеристики на КМ. Топологии на КМ – статични и динамични КМ. Архитектура на комуникационния елемент. Примери на КМ.

9. Архитектура на паметта в паралелните компютри.

Вертикална (йерархична) организация на паметта. Хоризонтална организация на паметта. Работа на кеш паметта.

10. Архитектура на дисковата памет.

Въведение. Характеристики на работното натоварване. Дискови матрици. Масиви от дискове с излишък – основни положения, нива на RAID. Сравнение между различните нива. Архитектура на контролерите. Архитектура на дисковата памет в компютрите с разпределена памет.

11. Производителност на компютъра. Методи за определяне на производителността.

Производителност. Видове производителност. Определяне на производителността – методи на измерването и методи на моделирането. Точност на различните методи. Ефективност.

Фиг.1-1. Вертикална структура на архитектурата на конвенционален (последователен) компютър.

Класическите задачи, които трябва да се решат при разработването на архитектурата на компютъра, могат да се групират в следните три групи:

- да се определи формата за представяне на програмата на компютъра и правилата на интерпретация;
- да се установят методите за адресация на данните в тези програми;
- да се определят форматите на данните.

При решаването на всеки от посочените проблеми трябва да се решат задачите за определяне минималната област на паметта, типа и форматите на данните, кодовете на операциите и форматите на машинните команди, способите за адресация и защита на паметта, механизма за управление на последователността на изпълнение на командите, интерфейса на компютъра с устройствата за вход/изход.

Основните отличия засягащи архитектурата на съвременните компютри могат да се резюмират по следния начин:

- Широко използване на процесори с **RISC** архитектура;
- Масово използване на паралелна обработка на различни нива;
- Използване на шината като основна среда за трансфер на информацията в компютъра.

Нека да разгледаме накратко до какво водят тези особености в архитектурата на съвременните компютри.

- Всички съвременни процесори спадат към една от двете групи:

а) процесори със сложен набор команди (**C**omplex **I**nstruction **S**et **C**ode – **CISC**);

б) процесори със съкратен набор команди (**R**educed **I**nstruction **S**et **C**ode – **RISC**).

Поради факта, че архитектурата на процесорът дава облик на цялата архитектура на компютъра, терминът Code в горните съкращения се заменя с Computer.

Идеята при **RISC** архитектурата е да се постигне бързодействие, посредством осигуряване на минимално време за изпълнение на често използваните команди. Този подход, а също така и обстоятелството, че сложната система от команди изисква по-големи системни издръжки, води до система със съкратен набор команди.

За да бъдат два процесора функционално еквивалентни, единият от които е **CISC**, а другият **RISC**, е необходимо функциите на процесора, реализирани от рядко срещаните **CISC** команди да се заменят с последователност от **RISC** команди в **RISC** процесора.

Понастоящем няма единно определение за това, какво е **RISC** - архитектура. Все пак преобладава становището, че за "чистата" **RISC** - архитектура са характерни следните особености:

а) командите трябва да се изпълняват за един и същ брой тактове (в идеалния случай за един такт);

б) командите трябва да имат един и същ формат;

в) обръщението към паметта е свързано само с команди от тип LOAD и STORE;

г) всички аритметични и логически функции се изпълняват на ниво регистър.

Трябва да се подчертае, че не във всички **RISC** - процесори, по една или друга причина, се реализират и четирите изисквания едновременно. Най-често се нарушават изисквания а) и б).

• Втората особеност в архитектурата на съвременните компютри се явява масовото използване на различни нива на паралелна обработка и особено на конвейерна обработка. Според много специалисти, конвейерната обработка на данни и команди е естественото развитие и усъвършенстване на класическия процесор. Действително, конвейерна обработка за пръв път се реализира в компютъра Strech на IBM през 1960 г., а по-късно и в компютъра на CDC 6600, но масовото приложение на този подход за повишаване на производителността се дължи на разпространението на **VLSI** (**V**ery **L**arge **S**cale **I**ntegration) технологията.

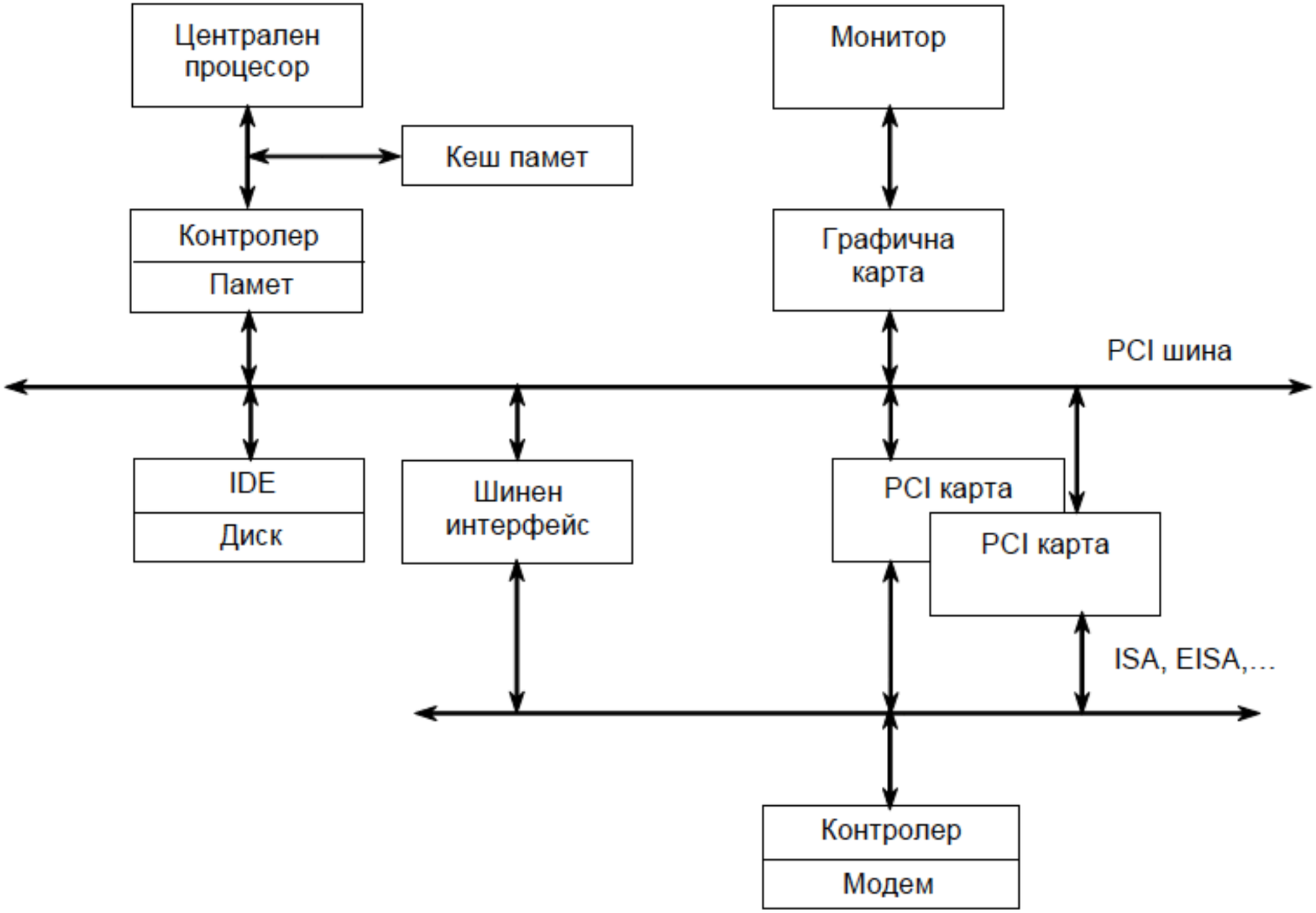
• И на края, като трета особеност се явява използването на шината като основно среда за трансфер на информацията в компютъра.

Шината е въведена като среда за пренасяне на данни и команди в средата на 60-те от DEC в компютъра PDP-8, като алтернатива на канала в големите машини. Поради своята простота и ниска цена тя намира масово приложение най-напред в микро- и мини- компютрите, а по-късно и в големите компютри. За да се отстрани най-сериозният недостатък на шината – ниската пропускателна способност, се използва:

а) увеличаване на ширината на шината;

б) увеличаване на тактовата честота;

б) разделяне на шината, позволяващо конкурентен достъп и възможност за извършване на множество транзакции едновременно, т.е. въвежда се йерархия от шини.



Фиг.2-1. Йерархична организация на шината

1. Необходимост от паралелна обработка

Исторически погледнато, развитието на архитектурата на компютрите е преследвало увеличаване на производителността и надеждността им. За постигането на тези цели се прилагат основно два подхода:

- Технологично усъвършенстване
- Организация на изчислителния процес.

Двата подхода не са алтернативни, а напротив, те взаимно се допълват, за което свидетелстват например съвременните персонални компютри, притежаващи изчислителни възможности надхвърлящи тези на компютрите преди 15 г. Но през първите 30 г. от развитието на компютрите основно се е "експлоатирал" първият подход (от ламповата технология до **VLSI**). Най-общо казано стремежът е бил към намаляване на продължителността на цикъла на процесора, напр. 500 ns при IBM 360/50 (1965 г.); 12,5 ns при CRAY-1 (1976 г.); при CRAY X/MP -9,5 ns (1982 г.) и т.н. Понастоящем максималната тактовата честота, която използват двата водещи производители на процесори Intel и AMD е от порядъка на 3 GHz. Не трябва да се отмине и разработвания от IBM нов компютърен дизайн "Interlocked Pipeline CMOS", който ще позволи на чиповете да достигнат скорости от 4.5 GHz

И така едно от предимствата на паралелните компютри спрямо конвенционалните е значително по-високата скорост на обработка, изразяваща се понастоящем в няколко GFLOPS.

Независимо от постигнатите в последно време скорости на обработка, има много области в които производителността на съвременните компютрите е поне на порядък по-ниска в сравнение с нуждите за ефективно решаване на приложни задачи. Такива "традиционни" области са геофизиката, молекулярната биология, моделирането на електронни схеми, а така също и при обработката на сигнали и изображения. В същото време високопроизводителните компютри намират все по-широко приложение в такива комерсиални области като финансовото моделиране и организацията на бази данни.

Какви са другите предимства на паралелните компютри?

- Повишена надеждност на компютъра. Тука компютърната надеждност може да се реши по друг начин, а именно като дефектиралия процесор се изключи от обработката, като същевременно тя продължава, естествено с намалена скорост.
- По-високо отношение производителност/цена. Една интегрирана оценка за всеки компютър е отношението производителност/цена, независимо от значителните понякога изменения на цените на един и същ компютър, дължащи се на маркетинговата политика на фирмата-производител. Като правило това съотношение е на порядък по-високо за паралелните компютри в сравнение с последователните.

Паралелна обработка в компютрите е възможно да се реализира на следните нива:

- На ниво задания. Тука паралелизъм е възможно да се реализира на две поднива.

а) Между заданията.

б) Между фазите на заданията.

Това е една от най-рано възникналите и експлоатирани форми на паралелизъм. За реализацията са необходими няколко паралелно работещи компютъра, несвързани помежду си или слабо свързани, разположени в едно помещение или в съседни помещения. Това е така наречената многомашинна система.

Паралелизъм от този тип представлява интерес по-скоро за системните администратори, отколкото за обикновения потребител.

- На ниво програми. Тука паралелизъм е възможно да се реализира също на две поднива.

а) Между частите на програмите (подпрограмите).

б) В границите на оператора за цикъл.

• На ниво команди. Това е възможно да стане между фазите на изпълнението на командите. Тази паралелност се постига чрез конвейерна обработка на командите и е паралелизъм от ниско ниво.

• На ниво машинна дума и аритметични операции. Тука паралелизъм е възможно да се реализира също на две поднива:

а) Между елементите на векторните операции. Този паралелизъм е характерен за векторните компютри, където наред с другите прийоми за увеличаване на производителността се използват и операционни (аритметични) конвейери.

б) Вътре в логическите схеми на аритметичното устройство. На практика всички процесори, дори и тези в еднопрцесорните компютри реализират паралелизъм на това подниво. От тази гледна точка, чисто последователен компютър няма, защото той би работил изключително бавно.

И в този случай реализирания паралелизъм е от ниско ниво.

След въвеждане на паралелна обработка на някое от нивата разгледани по-горе, е интересно да се оцени с колко се е променила производителността на новополучения компютър. За оценка на тази промяна се предлага простата формула

$$S = \frac{T_1}{T_N} \tag{3.1}$$

където: S е коефициент на изменение на бързодействието;

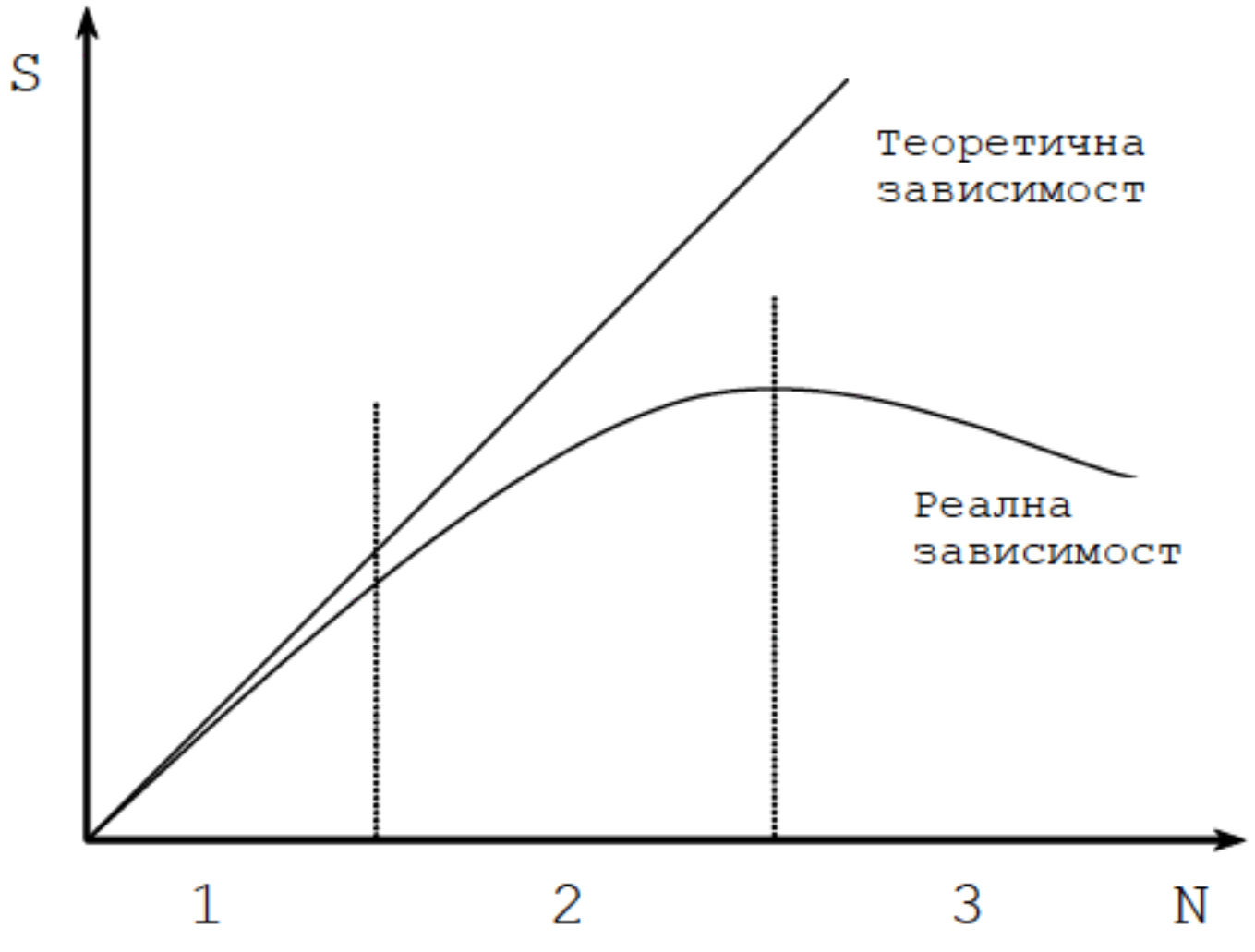
T_1 е времето за решаване на дадена задача на еднопроцесорен компютър;

T_N е времето за решаване на същата задача на паралелен компютър с N на брой процесора, имащи <u>същите характеристики както тези на процесорът от еднопроцесорния компютър</u>

За да има смисъл от въвеждането на паралелна обработка, очевидно е, че трябва да е в сила отношението **S>1**. Формула (3.1) показва относителното изменение на производителността на компютъра, след въвеждане на паралелна обработка. Така, този коефициент <u>отчита архитектурните особености</u> и не зависи от технологията на производство.

Ефективността от въвеждането на паралелна обработка се дава чрез:

$$E = \frac{S}{N} \leq 1 \tag{3.2}$$

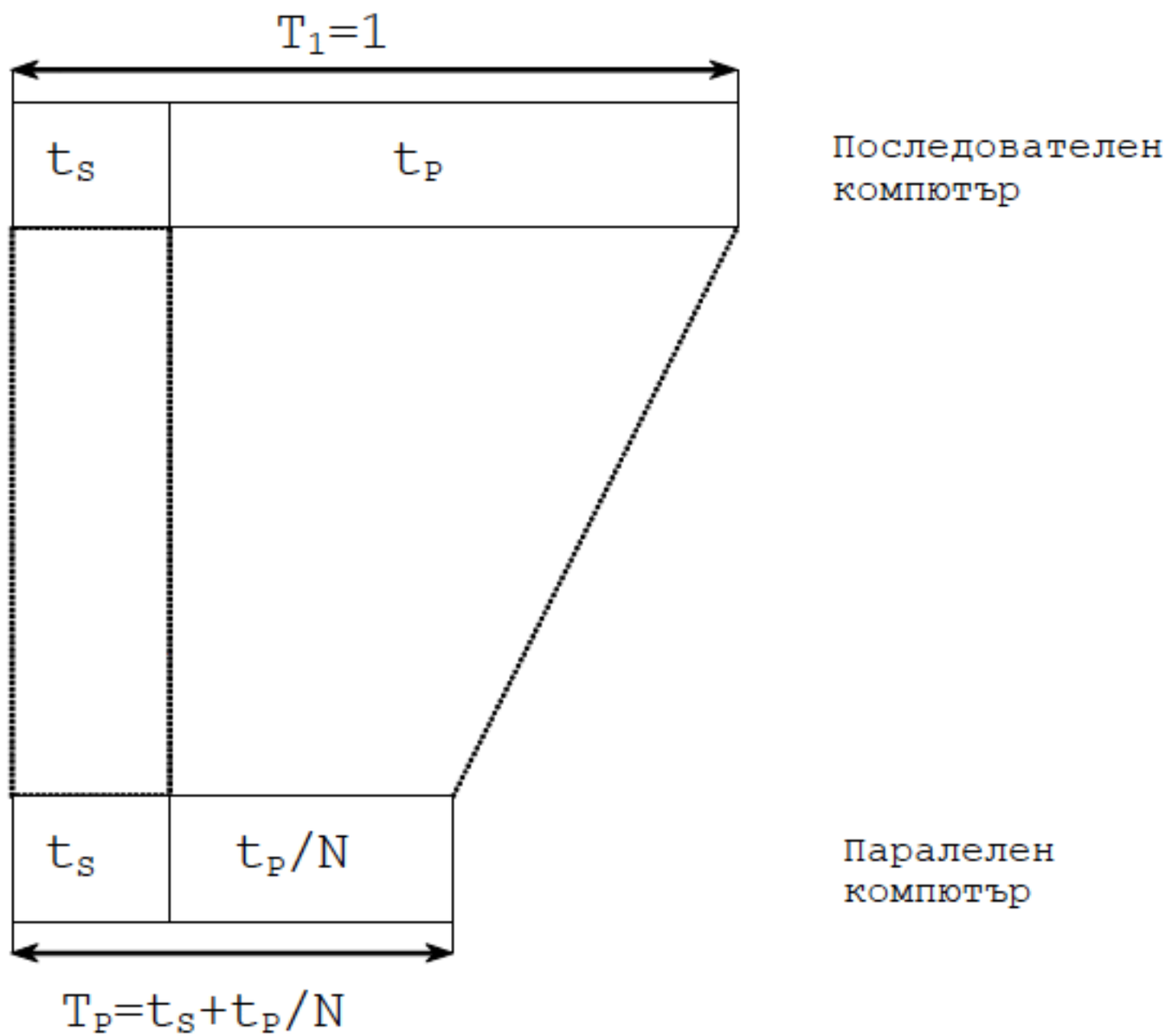


Фиг.3-1. Типична зависимост на производителността на паралелен компютър от броя на процесорите

През 1967 г. Amdahl предлага своя закон гласящ, че последователната част от задачата ограничава отгоре коефициента **S**. При това той привежда следните разсъждения. В една задача, решавана на еднопроцесорна система, има част, която може да се реши само последователно (обикновено тази част е свързана с координацията на целия изчислителен процес), а друга част – паралелно. Нека $\mathbf{t_s}$ е времето изразходвано за работа по последователната част, а $\mathbf{t_p}$ е времето изразходвано за работа по паралелната част, при това с цел алгебрично опростяване полагаме $\mathbf{t_s+t_p=1}$. Тогава, прилагайки формула (3.1) се получава

$$S = \frac{T_1}{T_N} = \frac{t_s + t_p}{t_s + \frac{t_p}{N}} = \frac{1}{t_s + \frac{t_p}{N}} .$$

Ясно е, че с увеличаване на **N** коефициентът **S** асимтотически се стреми към $\mathbf{1/t_s}$ – (виж фиг.3-2) и следователно последователната обработка, присъща на задачата, ограничава производителността на паралелния компютър.



Фиг.3-2. Модел на производителността при фиксиран размер на задачата

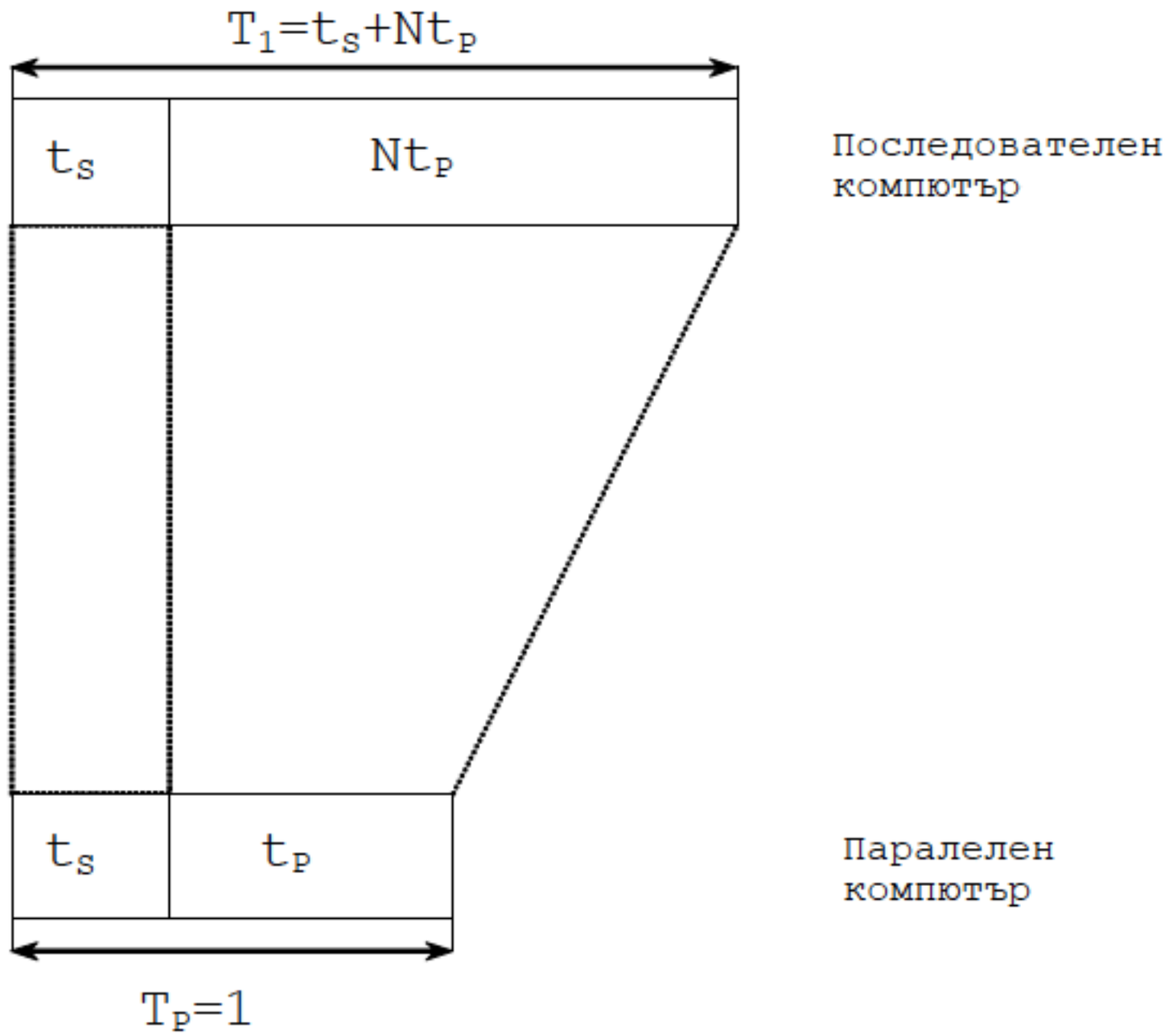
През 1987 г. Barsis предлага алтернатива на закона на Amdahl.

При закона на Amdahl се предполага, че $\mathbf{t_p}$ е независимо от **N**, което никога не е изпълнено. На практика програмистът може да варира с броя стъпки по време, стъпки в мрежата, точност на решението и други параметри, с които да настрои паралелната програма за изпълнение за предварително определено време. Следователно много по-реалистично е да се приеме, че времето за изпълнение е константа, отколкото размерът на задачата.

Ако отново използваме означенията $\mathbf{t_s}$ и $\mathbf{t_p}$, но дефинираме $\mathbf{t_s+t_p=1}$ като време за изпълнение на задачата на паралелния компютър, то тогава времето за обработка на последователния ще бъде $\mathbf{t_s+Nt_p}$ - фиг.3-3 и за **S** се получава:

$$S = \frac{T_1}{T_N} = \frac{t_s + Nt_p}{t_s + t_p} = t_s + Nt_p \, .$$

По такъв начин коефициентът на бързодействие расте с увеличаване на процесорите почти линейно. Например, ако **P**=1024 и $\mathbf{t_s}$=0.01 (както в горният случай), то **S**=1013.77 и **E**=0.99.



Фиг.3-3. Модел на производителността при фиксирано време

Флин през 1966 г. Класификацията на Флин се базира не на структурата на компютъра, а на това как в компютъра командите се обвързват с обработката на данните. Флин въвежда понятието поток и го дефинира така: последователност от елементи (данни и команди) изпълнявани или обработвани от процесорите. Във всеки компютър има два основни потока - единият от команди, а другият от данни. При срещата им се извършва обработка над данните.

В съответствие с това дали потоците от данни или команди са единични или представляват множество, възникват и следните четири големи класа:

SISD (**S**ingle **I**nstruction Stream, **S**ingle **D**ata Stream).

В този клас влизат обикновените фон Нойманoвски компютри, които имат един поток команди и един поток данни, т.е. едно устройство за обработка на командите и едно изпълнително устройство.

SIMD (**S**ingle **I**nstruction Stream, **M**ultiple **D**ata Stream).

В тези компютри се съхранява един поток от команди, но вече той е векторен, който инициира многочислени операции. Всеки елемент на вектора се разглежда като елемент на отделен поток от данни. В този клас се включват всички компютри с векторни команди, например CRAY-1, CRAY-2, CRAY X-MP, CRAY CS 6400, ILLIAC-IV, DAP на ICL, OMEN -64 и т.н.

Флин посочва следните проблеми, възникващи при разработката на **SIMD** компютрите:

а) Поддържане на комуникация между изчислителните елементи (процесори).

б) Необходимост от съответствие между размера на вектора от данни и размера на масива от процесори, който ще обработва този вектор.

в) Наличие на не векторни операции в програмите и появяване на допълнителна работа, свързана с подготовка за изпълнение на векторни операции.

г) Бездействие на голяма част от процесорите при наличие на условни преходи в програмата.

MISD (**M**ultiple **I**nstruction Stream, **S**ingle **D**ata Stream).

Към този клас компютри се отнасят компютри с множество потоци от команди и един поток от данни. Единствената архитектура, която с някакво основание може да бъде отнесена към този клас е конвейерната и то при условие, че всеки етап от изпълнението на командата се счита за една отделна команда. Но много по-естествено е да се отнесат конвейерите към класа **SIMD**. Действително самият конвейер може да се счита като система с архитектура **MIMD**, а по отношение на векторните операции той напомня архитектурата **SIMD**.

MIMD (**M**ultiple **I**nstruction Stream, **M**ultiple **D**ata Stream).

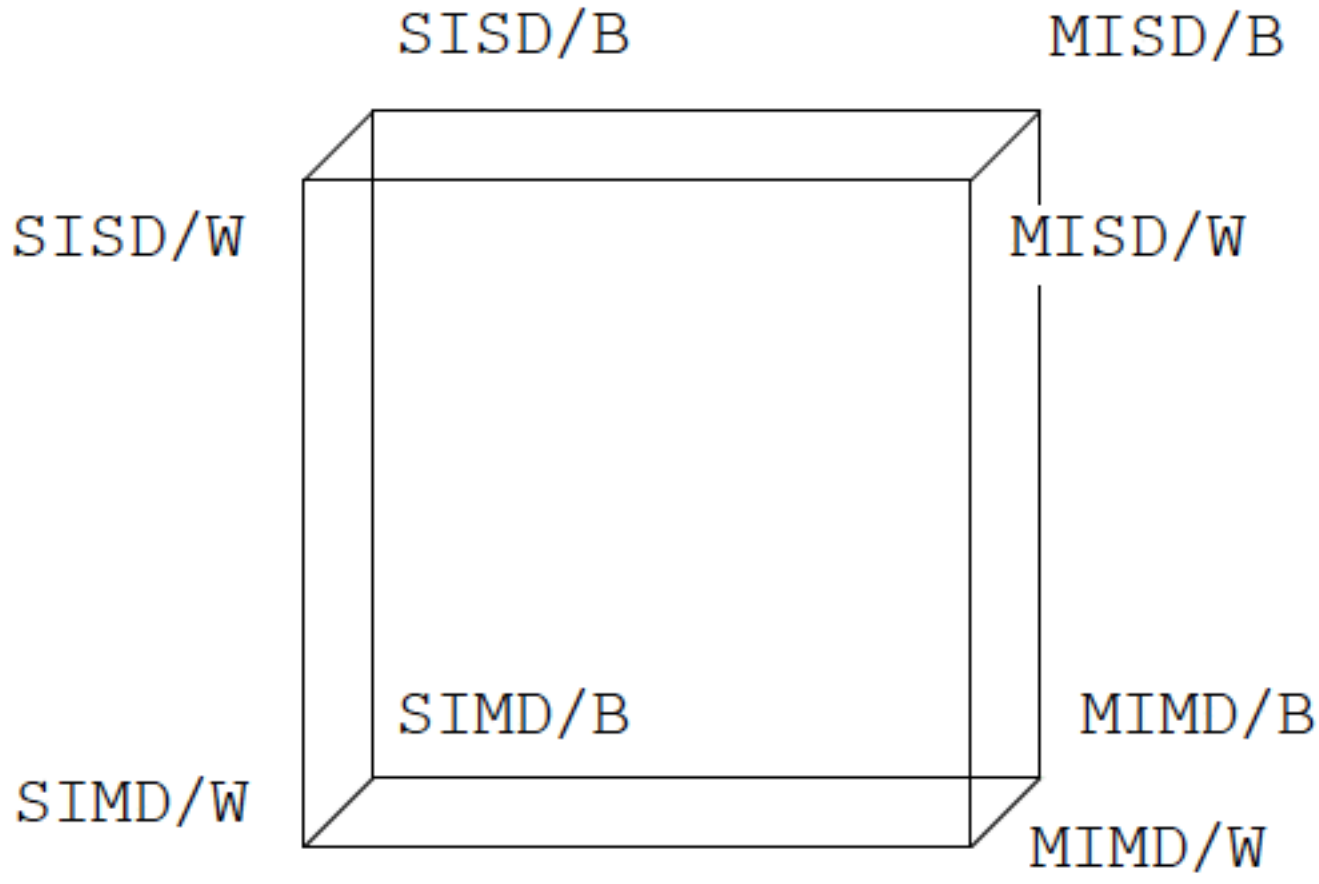
Множеството потоци от команди означава съществуването на няколко устройства за обработка на командите в този клас компютри. Затова тука се включват всички форми на мултипроцесорни конфигурации – от обединени компютри до матрици от процесори. В конфигурациите са включени няколко независими и завършени (пълноценни) еднопроцесорни компютри, които в процеса на решаване на задачите контактуват помежду си чрез съобщения или използват обща памет. Тъй като всеки от процесорите има свое управляващо устройство и може да изпълнява собствена програма, то за даден момент от време отделните процесори имат възможност да изпълняват различни операции върху различни данни.

Следните проблеми са общи за всички **MIMD** компютри:

а) Необходимост от препращане на данни и команди между процесорите. Тези операции обикновено поглъщат много време и за тяхното осъществяване е необходима (в повечето случаи сложна и скъпо струваща) комуникационна мрежа.

б) Цената се повишава линейно (в най-добрия случай) с увеличаване на броя на процесорите, докато скоростта на обработка се повишава много по-бавно поради увеличаване на броя на конфликтите между процесорите, използващи общи ресурси.

в) Осигуряване на методи за динамична реконфигурация на ресурсите на системата за задоволяване на постоянно променящите се изисквания и заетост на процесорите.



Фиг.3-4.Разширена класификация на Флин, отчитащ начина на обработка - по думи (W) или по битове (B)

Система на Флин	Тип автономия	Топология на мрежата	Формат на данните	Примери
MIMD		многостъпална	с ПЗ	PASM
		едностъпална		
		хиперкуб	с ПЗ	NCUBE: iPSE
		шина	смесен	DATACUBE
		линийка	с ПЗ	WARP
		кръг	с ФЗ	ZMOB
		решетка	с ПЗ	Victor
MISD		линийка	бит последователна	Цитокомпютър
		линийка	с ПЗ	PIPE
SIMD		решетка	бит последователна	CLIP 4, MPP, DAP, AAP, GRIP
		дървовидна	с ФЗ	NON VON
		тримерен куб	бит последователна	
	адресна	решетка	с ФЗ (8/16/32)	ILLIAC IV
	адресна	линийка	с ФЗ	CLIP 7
	адресна	многостъпална	с ПЗ	GF11, PASM
	мрежова	n куб		
	мрежова	полиморфна		YU PPIE
	операционна	пирамидална	бит последователна	PAPIA
	операционна	линийка	с фЗ	CLIP 5

Операционна автономия.

Осигурява в архитектура с масов паралелизъм (виж тема 8) възможност за изпълнение няколко различни операции върху мрежа от процесори. Това съвсем не означава, че всеки процесор работи по своя собствена програма, както при компютрите от тип **MIMD;** просто това означава, че не всички процесори изпълняват една и съща команда както е в традиционните **SIMD** компютри. Това се постига чрез:

• Техниката на маската. Чрез техниката на маската се реализират преходи в паралелната програма. Този въпрос е разгледан по-подробно в тема 8.

• Освен бит за активност (както е в CLIP7) се използва многоразряден регистър, който може да се разглежда като регистър на разширения код на операцията. Благодарение на това, всяка команда се интерпретира по-различен начин от всеки процесор.

• В паралелните компютри **MSIMD** (**Multiple SIMD**) има няколко блока за програмно управление (програмни контролери), всеки от който е свързан с определена група процесори (кластер). Такъв тип операционна автономия е реализиран, напр. в йерархичните архитектури с пирамидална топология (виж тема 8). Всяко ниво се управлява от отделен контролер, благодарение на което процесорите от едно ниво работят в режим **SIMD**, но на различни нива се изпълняват различни програми.

Адресна автономия.

Осигурява на всеки процесор в **SIMD** компютъра възможност за формиране адреса локално или да модифицира получения адрес, независимо от другите процесори. Така може да избере операнд от собствената памет по адрес, отличен от адреса на другите процесори. Тази автономия отдавна е призната за важна, защото тя същевременно облекчава програмирането на **SIMD** компютрите, трудността при които се заключава в необходимостта от обръщение на всички процесори към една и съща клетка от паметта при изпълнение на една и съща команда.

Мрежова автономия.

Това е принцип на организация на изчисленията, имаща за цел ефективно отразяване на графа на задачата (породен от даден алгоритъм за изпълнение) в структурата на връзките между процесорите. При това се преследват същите цели, както и в компютрите с реконфигуруема комуникационна мрежа (виж тема 12), а именно осигуряване висок коефициент на използване на ресурсите.

- По функционално предназначение се отделят две направления на развитие:

а) Създаване на компютри за изпълнение на различни инженерни разчети, основно ориентирани за бързо изпълнение на аритметични операции.

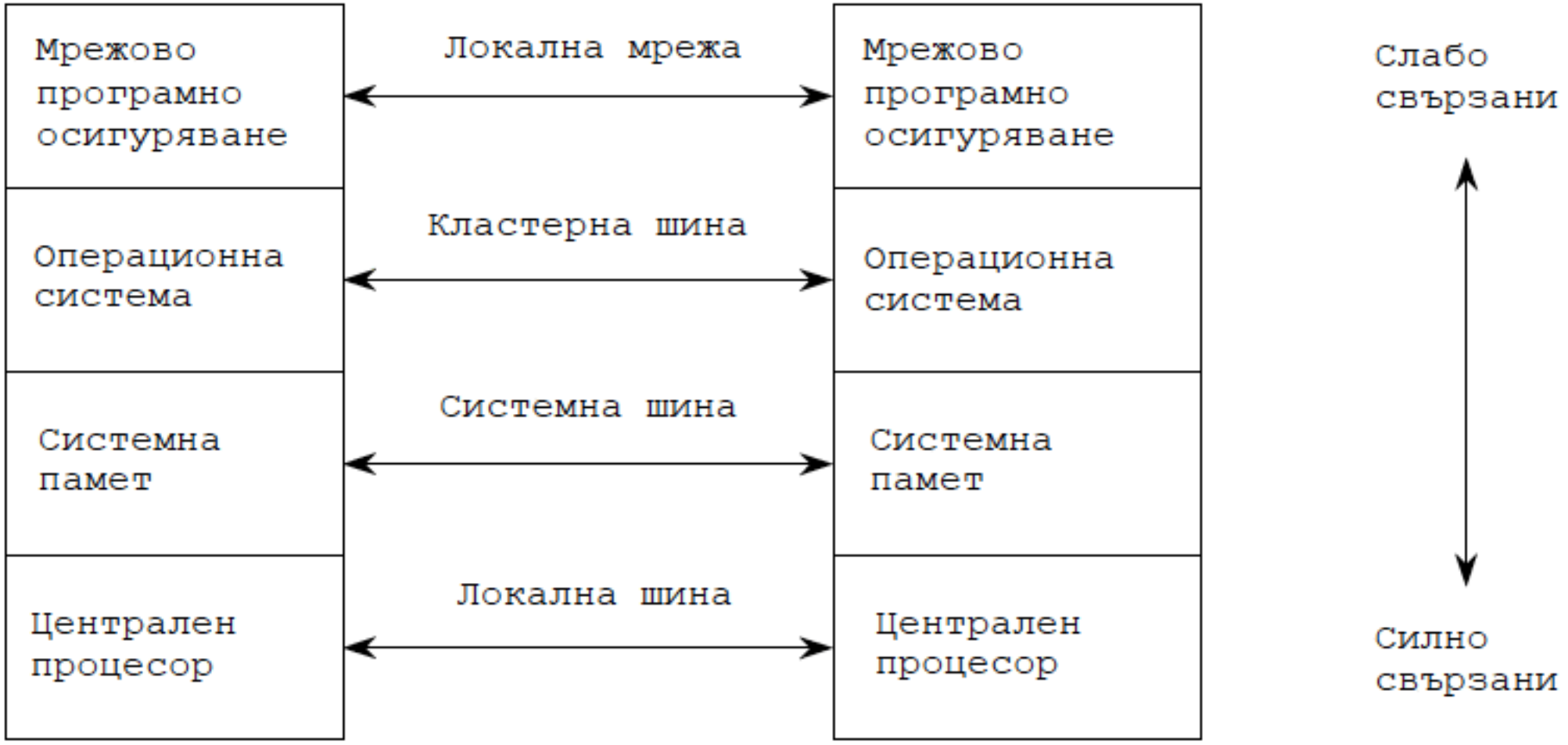
б) Създаване на компютри за системи с изкуствен интелект и бази знания.

- По способа за управление на паралелната работа се отделят три класа компютри:

а) С управление по потока управляващи оператори (control driven).

б) С управление по потока данни (data driven).

в) С управление по потока заявки (demand driven).



Фиг.3-5. Класификация по степен на свързаност

- Класификация по степените на равноправие на процесорите е друга възможна класификация. В "автократичните" системи един от процесорите изпълнява ролята на управляващ (главен), а останалите – спомагателни функции. В "егалитарните" системи всички процесори са равноправни и всеки от тях може да започва изпълнението на нов, активен процес.

- Съществуват също така класификации и по способа на синхронизация на процесорите. Необходимо е да се отбележи, че синхронизацията може да бъде на различни нива – напр. на ниво команди или на ниво процедура и т.н.

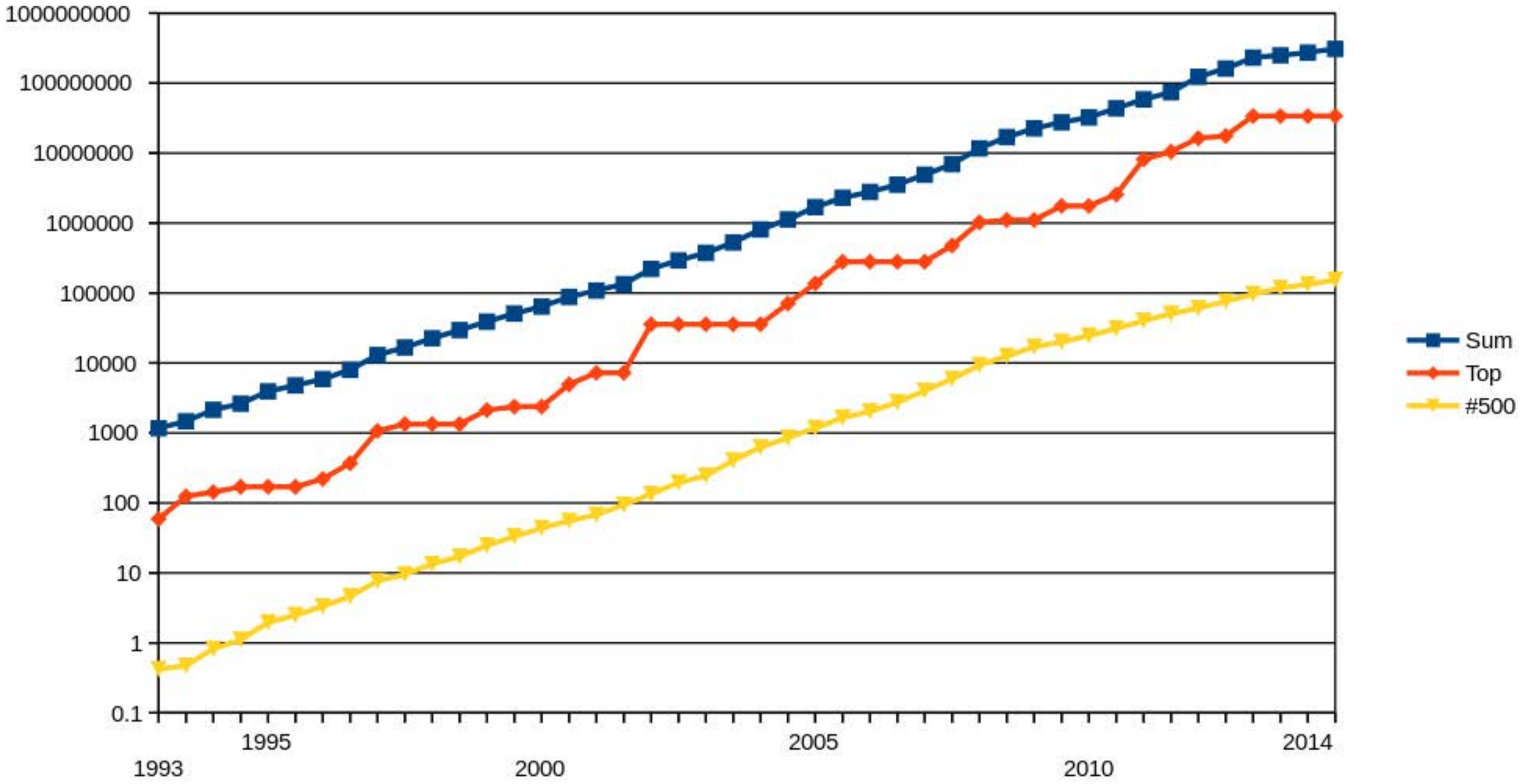
Кои са компютрите, достигнали максиамлна производителност през последните десетилятия и преминали следващата граница на производителността? Таблицата, поместена по-долу дава отговор на поставения въпрос.

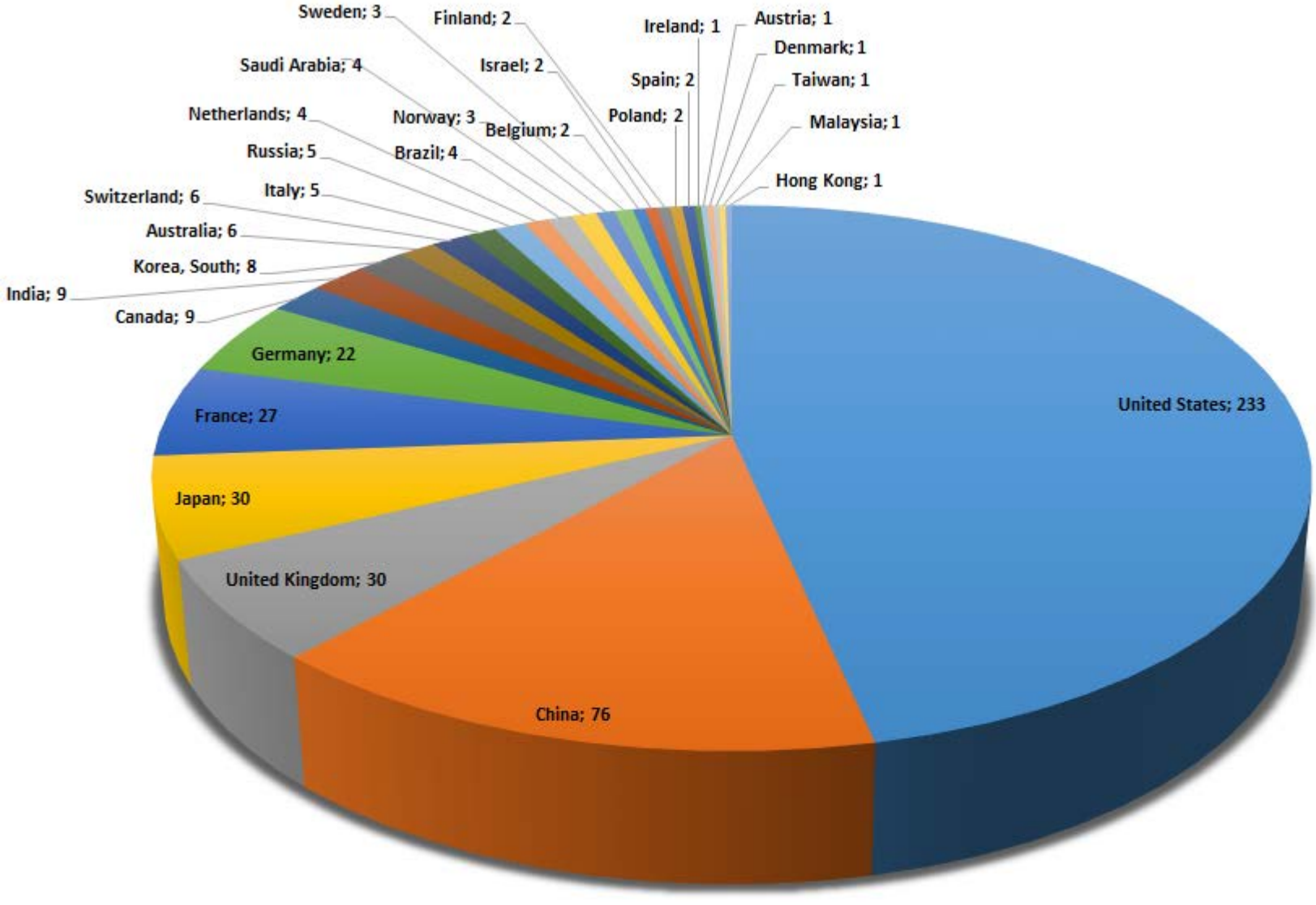
Година	Тип (Производител)	Граница на производителността	Производителност
1971	CDC 7600	MFLOPS	1.24 MFLOPS
1986	CRAY 2	GFLOPS	1.7 GFLOPS
1997	ASCI (Intel)	TFLOPS	1.068 TFLOPS
2008	Roadrunner (IBM)	PFLOPS	1.105 PFLOPS

На базата на тази таблица и вземайки предвид общите тенденции на развитието на технологиите се прави прогнозата, че всички компютри през 2015 влизащи в Top 500 ще бъдат с производителност по-голяма от 1 PFLOPS (10^{15}), а първият компютър с производителност от порядъка на 1 EFLOPS (10^{18})ще се появи през 2019 год.

Интересното е да се отбележи, че в началния период (края на 80-те и началото на 90-те години) голяма част от суперкомпютрите са базирани на векторната и **SIMD** архитектури. В стечение на времето, постепенно, компютрите със **SIMD** архитектура отпадат, а само 4 компютъра с вектрона аритектура (2 на CRAY и 2 на NEC) попадат сред най-бързите 30 компютъра. По настоящем разпределението по архитектури изглежда по следния начин:

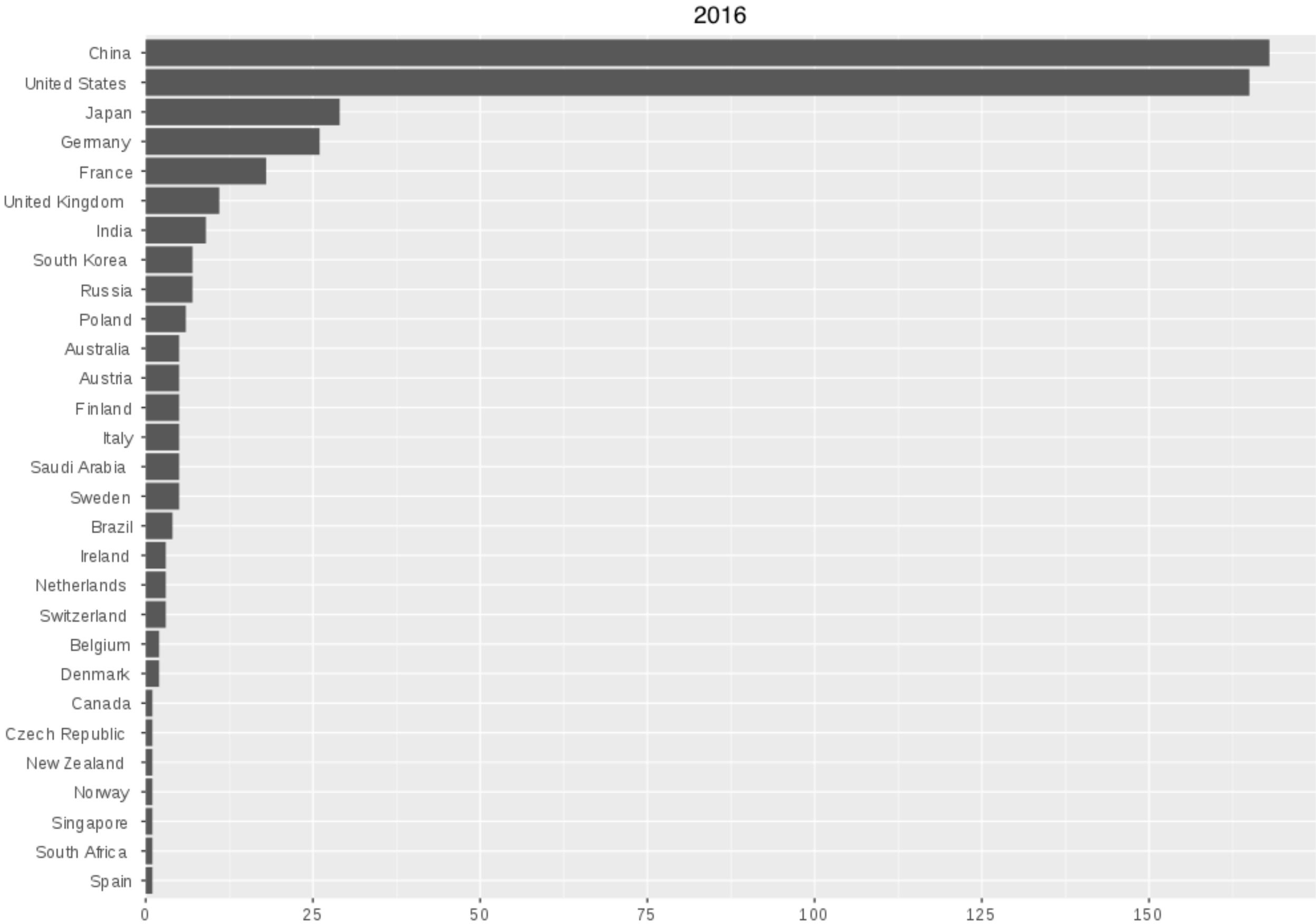
Архитектура	Процент
Скаларна	0 %
Векторна	2.8%
SIMD	0 %
SMP	0 %
MPP	16 %
MIMD (клъстерна)	81.2 %

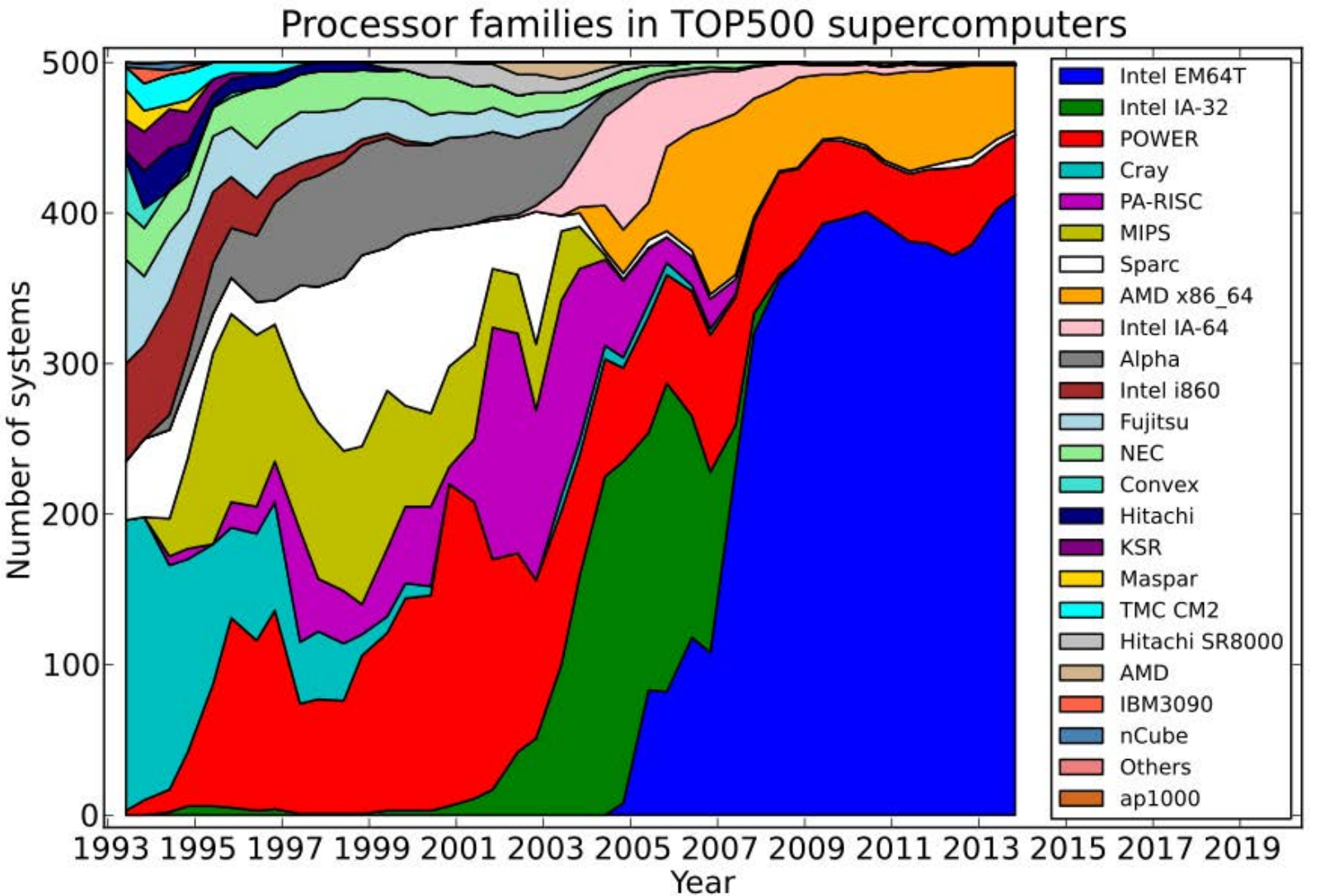

1000000000
100000000
10000000
1000000
100000
10000
1000
100
10
1
0.1
1993
1995
2000
2005
2010
2014
Sum
Top
#500

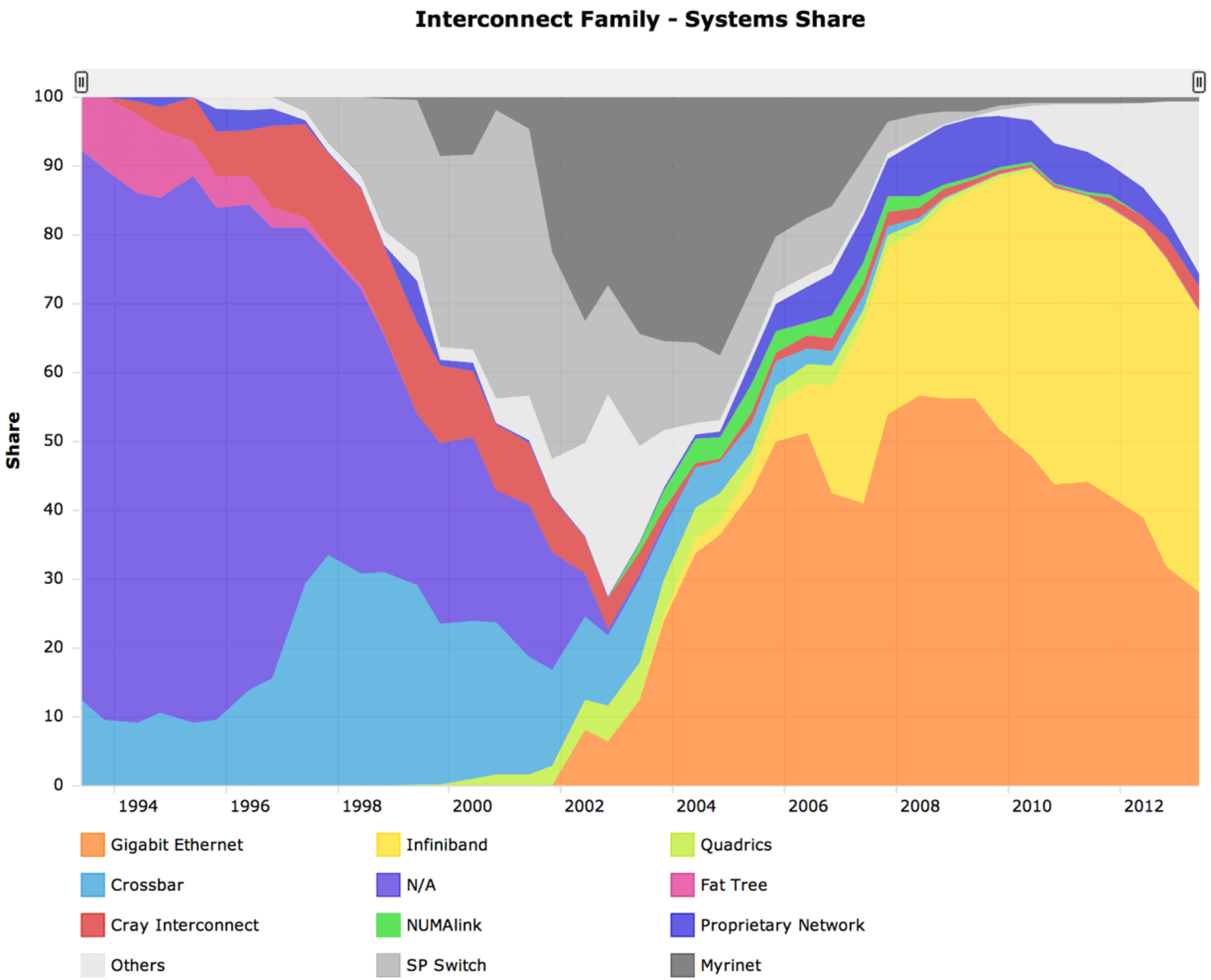


Supercomputer Share by Countries (June 2014)

Source: www.top500.org


2016
China
United States
Japan
Germany
France
United Kingdom
India
South Korea
Russia
Poland
Australia
Austria
Finland
Italy
Saudi Arabia
Sweden
Brazil
Ireland
Netherlands
Switzerland
Belgium
Denmark
Canada
Czech Republic
New Zealand
Norway
Singapore
South Africa
Spain
0
25
50
75
100
125
150


Processor families in TOP500 supercomputers
Number of systems
0
100
200
300
400
500
1993 1995 1997 1999 2001 2003 2005 2007 2009 2011 2013 2015 2017 2019
Year
Intel EM64T
Intel IA-32
POWER
Cray
PA-RISC
MIPS
Sparc
AMD x86_64
Intel IA-64
Alpha
Intel i860
Fujitsu
NEC
Convex
Hitachi
KSR
Maspar
TMC CM2
Hitachi SR8000
AMD
IBM3090
nCube
Others
ap1000


Interconnect Family - Systems Share
Share
100
90
80
70
60
50
40
30
20
10
0
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
Gigabit Ethernet
Infiniband
Quadrics
Crossbar
N/A
Fat Tree
Cray Interconnect
NUMAlink
Proprietary Network
Others
SP Switch
Myrinet

Site:	National Super Computer Center in Guangzhou
Manufacturer:	NUDT
Cores:	3,120,000
Linpack Performance (Rmax)	33,862.7 TFlop/s
Theoretical Peak (Rpeak)	54,902.4 TFlop/s
Nmax	9,960,000
Power:	17,808.00 kW
Memory:	1,024,000 GB
Processor:	Intel Xeon E5-2692v2 12C 2.2GHz
Interconnect:	TH Express-2
Operating System:	Kylin Linux
Compiler:	icc
Math Library:	Intel MKL-11.0.0
MPI:	MPICH2 with a customized GLEX channel

Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway

Site:	National Supercomputing Center in Wuxi
Manufacturer:	NRCPC
Cores:	10,649,600
Linpack Performance (Rmax)	93,014.6 TFlop/s
Theoretical Peak (Rpeak)	125,436 TFlop/s
Nmax	12,288,000
Power:	15,371.00 kW
Memory:	1,310,720 GB
Processor:	Sunway SW26010 260C 1.45GHz
Interconnect:	Sunway
Operating System:	Sunway RaiseOS 2.0.5

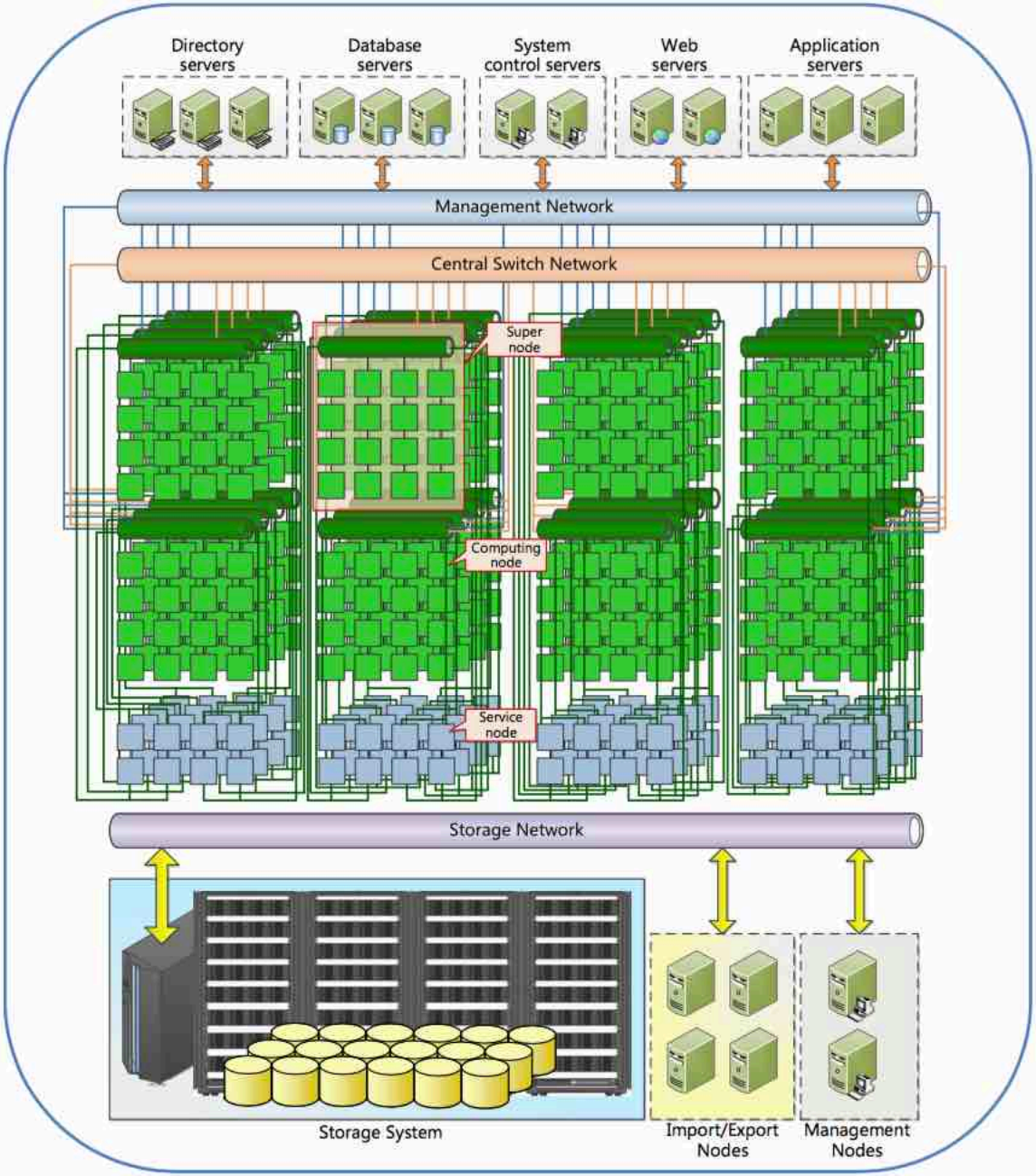


Figure 12: General Architecture of the Sunway TaihuLight


申威®
26010
HGJ ICDC
066802537580


神威
太湖之光
神威
SHENWEI TAIHU LIGHT

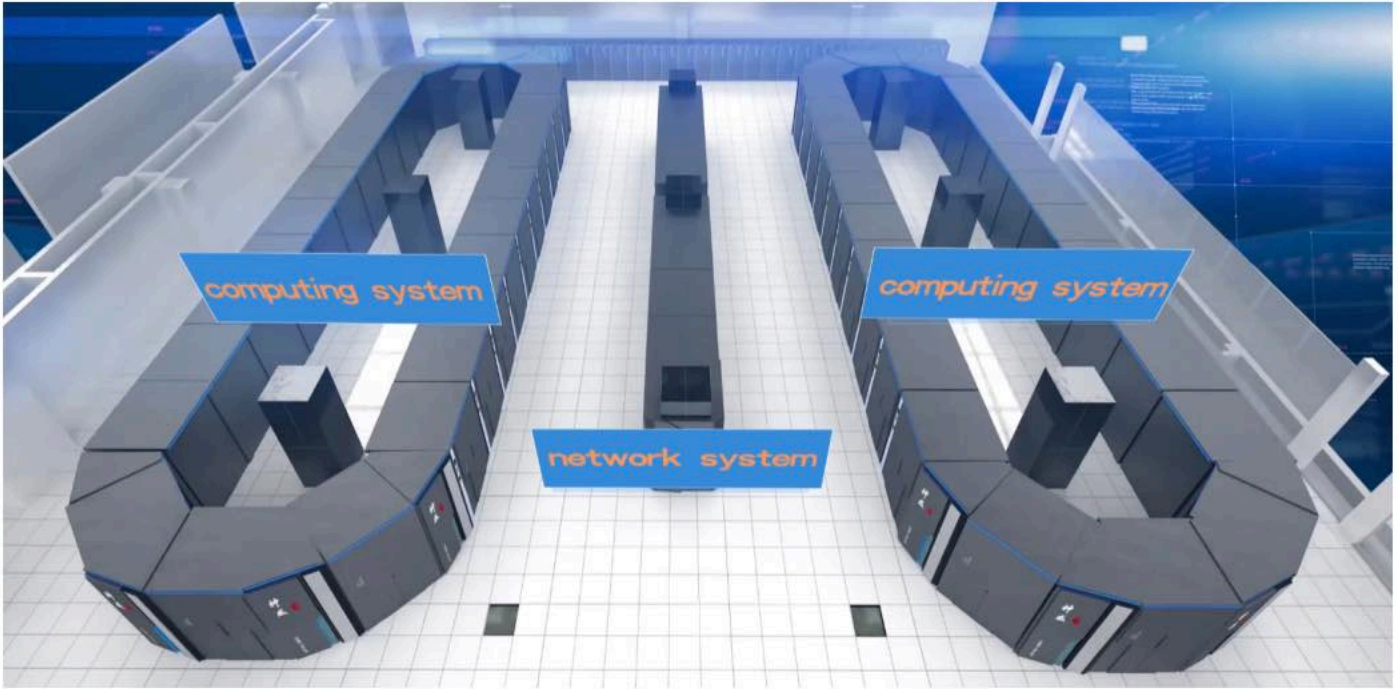
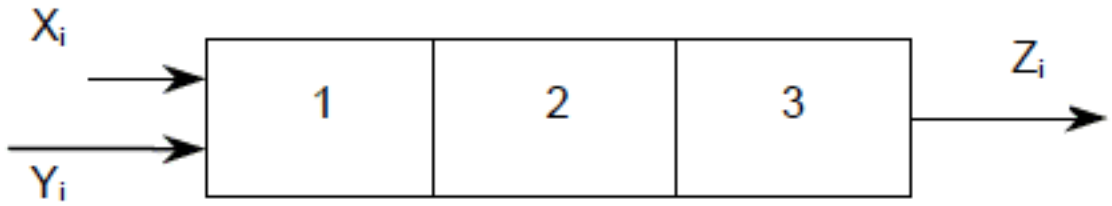


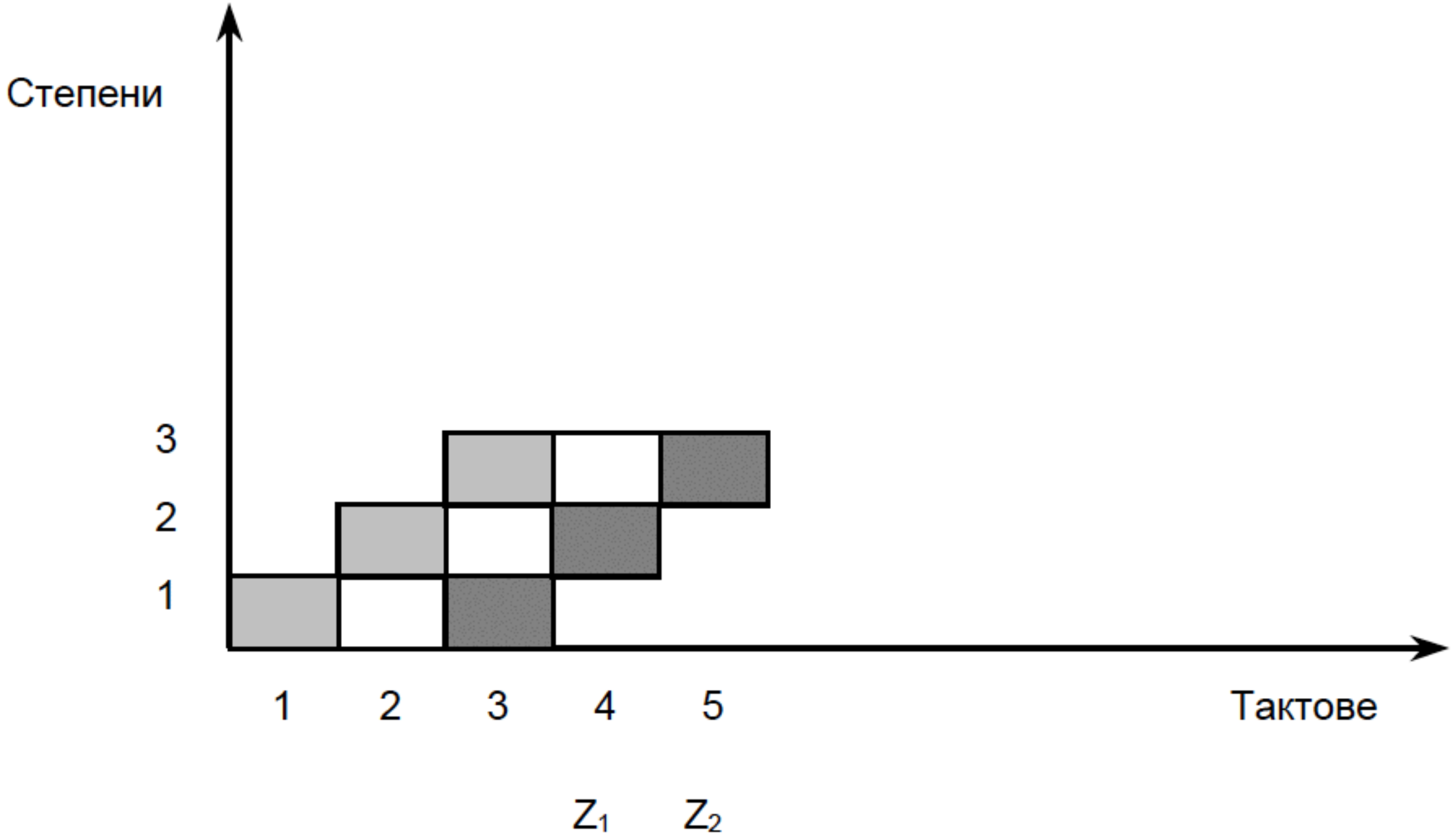
Figure 4: Overview of the Sunway TaihuLight System

Пример. *Да се разгледат процесите протичащи в един конвейер за събиране на два вектора с дължина* ***n*** *елемента. Форматът на данните са числа с плаваща запетая.*

Базовата функция (в случая събиране на числа с плаваща запетая) може да се разбие на следните подфункции*: а) сравнение на порядъците и изместване на мантисата на числото с по-малък порядък на дясно, на толкова разряда колкото е разликата между двата порядъка; б) сумиране на мантисите; в) нормализация на резултата. Тогава конвейерната обработка може да се реализира чрез 3 степенен конвейер - Фиг.4-1.



Фиг.4-1.Обобщена структурна схема на тристепенен конвейер.


Степени
3
2
1
1
2
3
4
5
Тактове
Z1
Z2

Работата на конвейера протича по следния начин:

1-ви такт. Първата двойка числа постъпват за обработка в първата степен на конвейера, където се реализират операциите по подфункция а).

2-ри такт. Получените междинни резултати от първата степен се предават на втората степен, която извършва операциите по реализация на подфункция б). В същото това време първата степен е свободна и може да поеме обработката на следващите двойки числа.

3-ти такт. По времетраенето на третият такт, в третата степен окончателно се формира резултатът за първата двойка числа. Същевременно, първата степен работи по третата двойка числа, а втората е заангажирана с работата по втората двойка числа.

4-ти такт. През време на четвъртия такт, първият резултат напуска конвейера. Третата степен работи по подфункция в) за втората двойка числа; втората степен работи по третата двойка числа; първата степен работи по четвъртата двойка числа.

5-ти такт и т.н.

След като се запълни конвейерът на всеки такт се генерира по един резултат. Така за обработката на всичките **n** елемента на векторите са необходими **n+2** такта.

За оценка на производителността се изхожда от основното съотношение (формула 3.1).

$$S = \frac{T_1}{T_P},$$

където $\mathbf{T_1}$ в случая е брой тактове при последователните изчисления; $\mathbf{T_P}$ е брой тактове при конвейерните изчисления.

Така за случая се получава:

$$S = \frac{3n}{3+(n-1)} = \frac{3n}{n+2}.$$

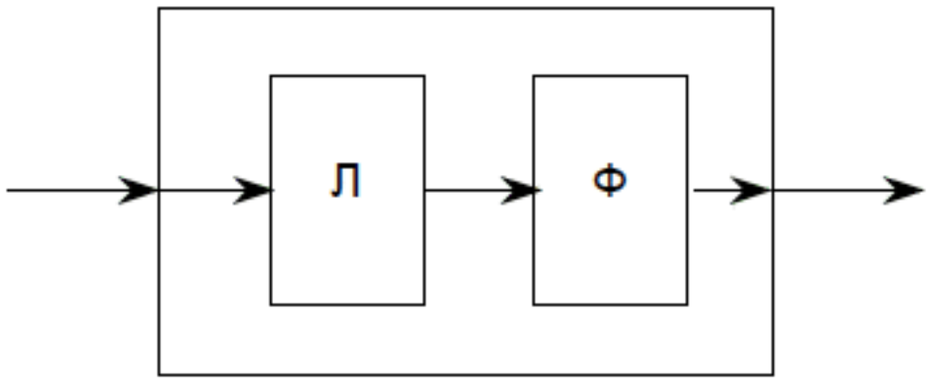
Или обобщено за конвейер с **L** степени:

$$S = \frac{Ln}{L+n-1}.$$

Конвейерност означава, че се осигурява съвместяване във времето на различни действия по изчисляването на базовите функции за сметка на тяхното разбиване на подфункции. За да се реализира конвейерна обработка е необходимо да са изпълнени следните пет условия:

- Изчислението на базова функция е еквивалентно на някаква последователност от изчисление на подфункции.
- Величините явяващи се входни за дадена подфункция, се явяват изходни за предходната подфункция.
- Никакви други взаимодействия между подфункциите няма освен входните и изходните величини.
- Всяка подфункция се реализира чрез апаратни блокове.
- Действията, реализирани от тези апаратни блокове изискват приблизително едно и също време.

Апаратните средства, необходими за изпълнението на всяка от тези функции, образуват степен. Всяка степен от своя страна е изградена от два блока: логика - (Л) и фиксатор -(Ф)- Фиг.4-3.



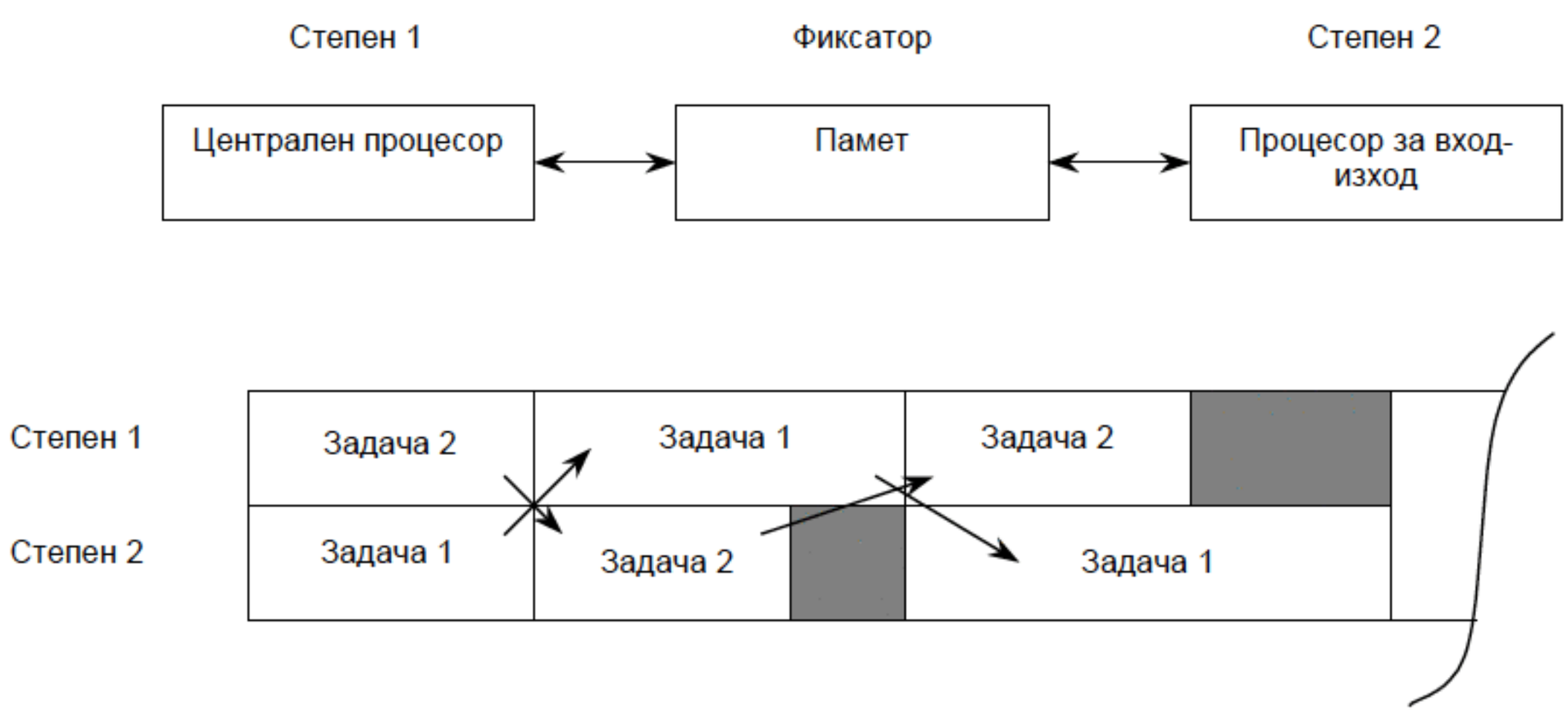
• между отделните обработки може да има някаква зависимост освен предаването (получаването) на данните;

• за всяка обработка може да е необходима различна верига от подфункции;

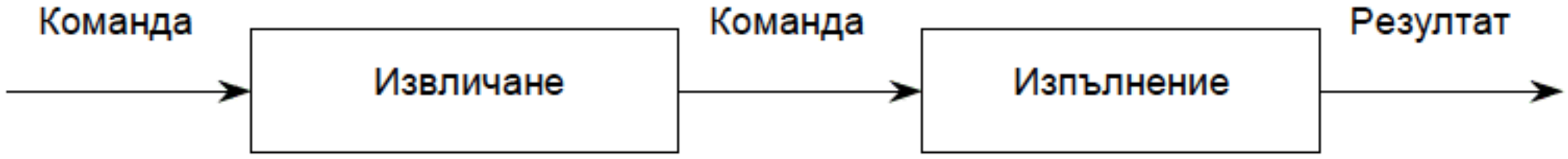
• по своето назначение подфункциите са достатъчно различни;

• времето, необходимо за всяка степен, не е обезателно постоянно.

Типичен пример в това отношение е препокриването на входно-изходните операции с изчислителните – Фиг.4-4.



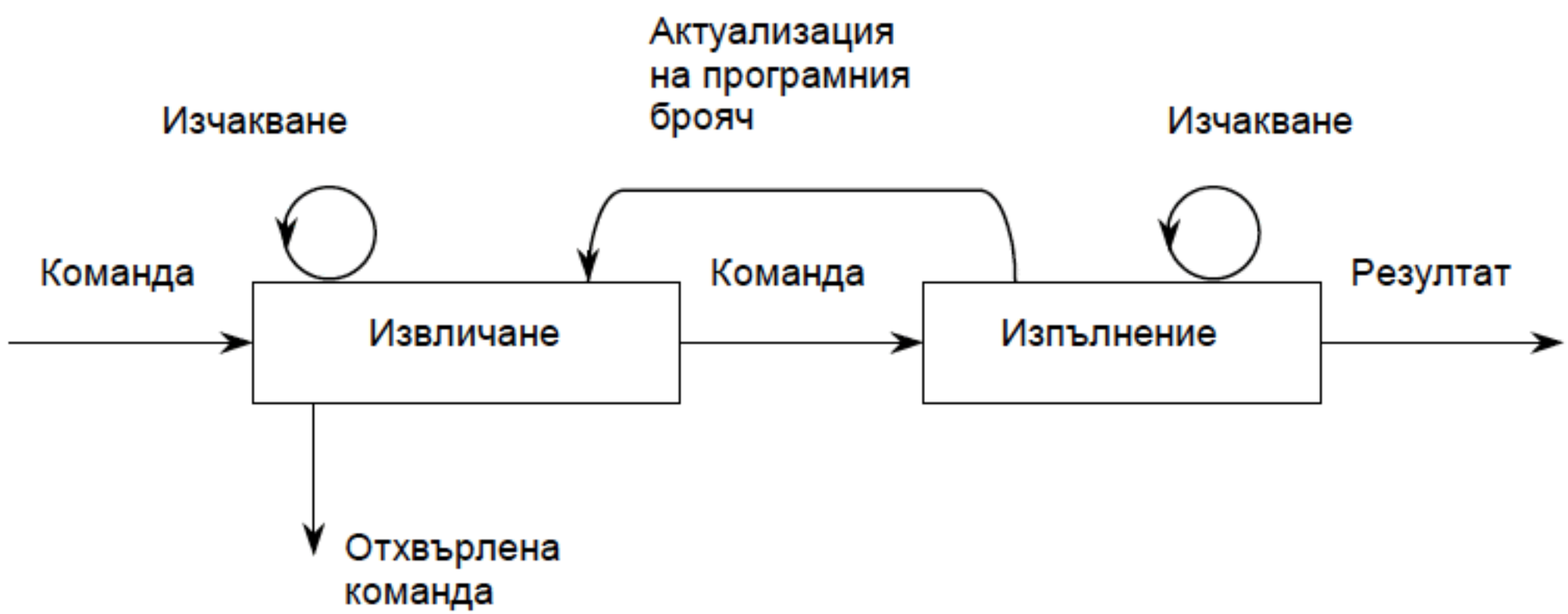
Известно е, че изпълнението на командите включва в себе си няколко етапа. В най-простия случай са два: извличане и изпълнение. В етапа изпълнение има интервали от време, когато обръщения към паметта отсъстват. Тези интервали могат да бъдат използувани за избор на следващите команди. Този подход е илюстриран на фиг.4-5.



Фиг.4-5. Възможно най-простата структура на конвейер за команди

Ако двата етапа имат еднаква продължителност, то времето за изпълнение на командата ще бъде съкратено двойно. Но удвоеното увеличение на скоростта е малко вероятно по две причини:

- Времето за изпълнение на командата, като правило, е много по-голямо от времето за извличане.
- При изпълнение на командите за преход е неизвестно каква команда трябва да се избере като следваща от паметта. Затова на етапа на избор е необходимо да се чака докато текущата команда от втората степен се изпълни. След това на етапа на изпълнение настъпва принудително изчакване за избор на следващата команда от паметта. Това е илюстрирано на фиг.4-6.



Независимо от всичко, така разгледания конвейер няма голямо ускорение. За увеличаването му се използват следните два прийоми:

а) Конвейерът е двустепенен, но в първата степен се изпълняват повече действия по цялостното изпълнение на командата, като извличане, декодиране, определяне адреса на операнда. Така се осигурява приблизително еднакви времена за изпълнението на функциите и за двете степени. В този случай е прието първата степен на конвейера да се означава като **IF**-устройство (**I**nstruction **F**etch Unit), а втората степен като **E**-устройство (**E**xecution Unit). Между двете степени трябва да има фиксатор. Най-простата структура е регистър, който съхранява поредната команда за изпълнение. В този случай връзката е твърда. При запълване на този регистър **IF**-устройство преустановява работата си. По-голямо разпространение е получил фиксатор, съставен от набор от регистри (регистров файл). Така е възможно всяка от степените да работи относително независимо от скоростта на другата степен.

б) Конвейерът да има по-голям брой степени, всяка реализираща различен етап от обработката на командата, напр. извличане, декодиране, определяне адреса на операнда и т.н.

Но с увеличаване броят степени в конвейера не следва пропорционално увеличение на скоростта на обработка. Причините за това са:

- Във всяка степен на конвейера възникват допълнителни загуби на време, свързани с прехвърлянето на данни между фиксаторите и изпълнението на различни подготвителни функции.
- При увеличаване степените на конвейера стремително се увеличава обема на управляващата логика, необходима за отчитането зависимостите при обръщението към паметта и регистрите, а така също и за оптимизация при използването на конвейера.

В по-старата генерация процесори (напр. I8086) е използуван първият подход за изграждане на конвейера, докато в съвременните процесори се предпочита втория.

Проблемите при конвейерното изпълнение на командите ще бъдат изяснени като се разгледа как протича обработката един програмен фрагмент, записан на хипотетичен асемблерен език

Да разгледаме изпълнението на следния фрагмент:

```
.................
LOAD A,M1
LOAD B,M2
ADD A,B
STORE M3,A
JUMP X
.................
```

Тука **A** и **B** са регистри, **M1**, **M2**, **M3** са клетки от паметта, а **X** е етикет (адрес на клетка от паметта).

IF - избор на командата;

E - изпълнение на командата.

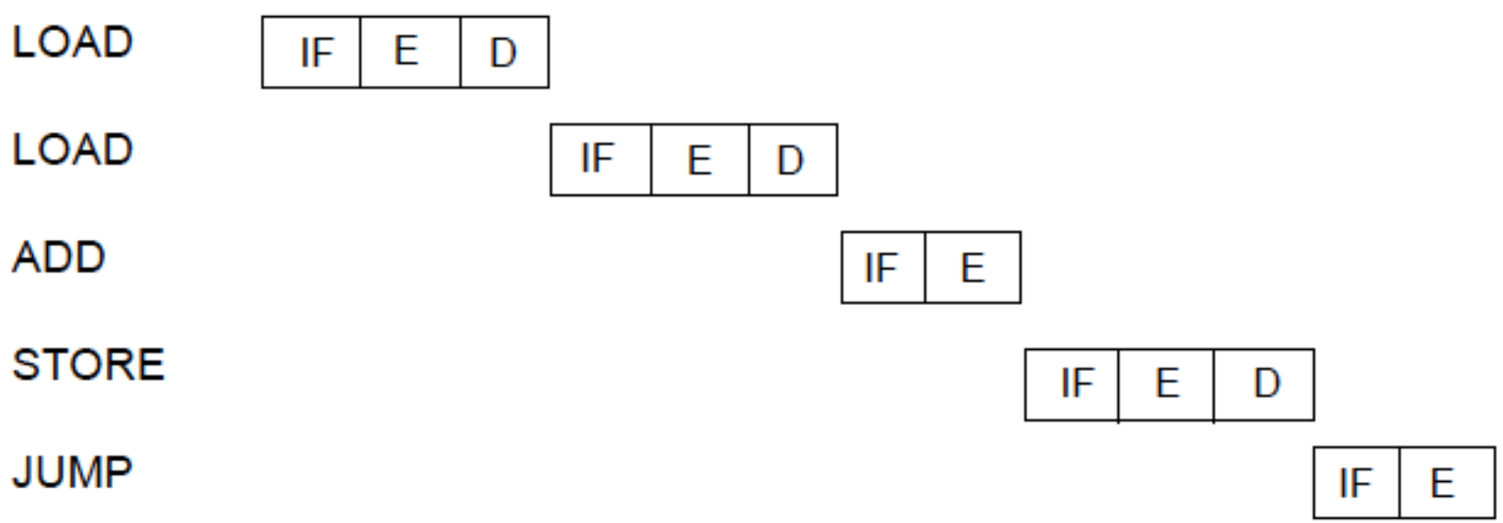
За изпълнението на операциите за четене/запис в/от паметта (**LOAD** и **STORE**) са необходими три фази:

IF - избор на командата;

E - изчисление на адреса на паметта;

D - обръщение към паметта.

На фиг.4-7 е показано "стандартното" т.е. не конвейерно изпълнение на програмния фрагмент.



Фиг.4-7. Не конвейерно изпълнение на примерна програма.

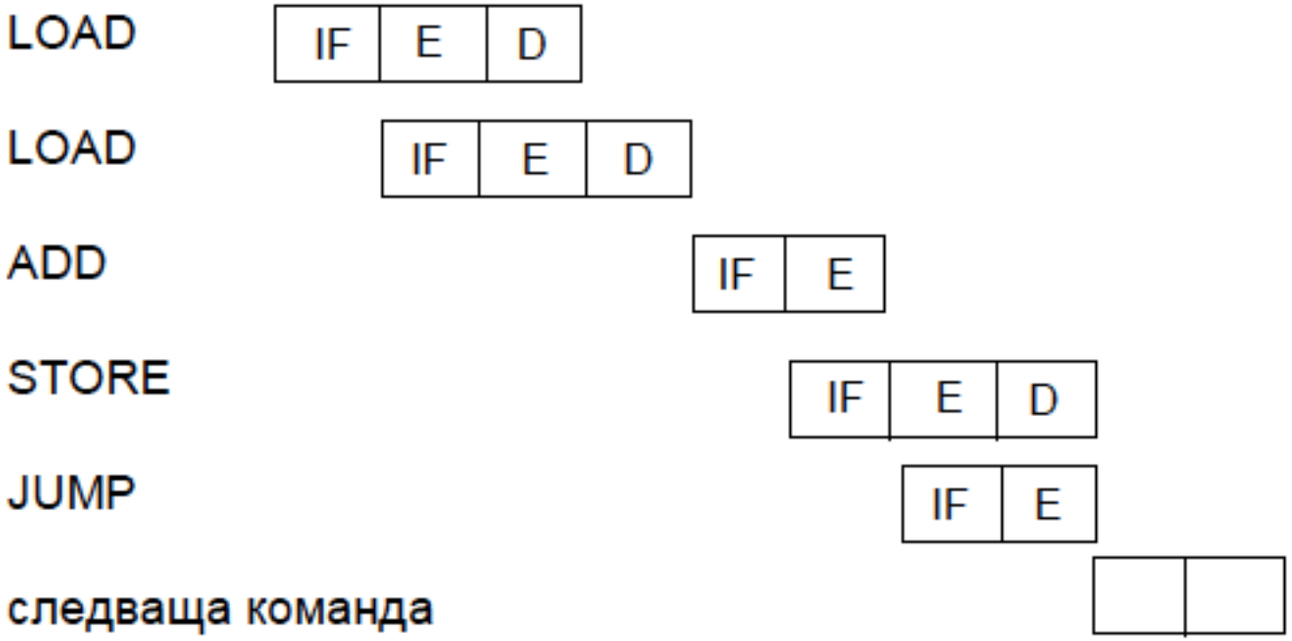
Лесно може да се установи, че изчислителният процес отнема 13 такта.

Преди да се обсъди конвейерното изпълнение на програмния фрагмент е необходимо да се конкретизира взаимодействието на процесора (конвейера) с паметта. Възможни са следните два случая:

- А) Процесорът има обща памет за данни и команди.
- Б) Процесорът има две памети – едната за данни, а другата за команди.

В зависимост от това са възможни два сценария за изпълнение. И за двата ще считаме, че за извличане на данните от паметта е необходим един цикъл.

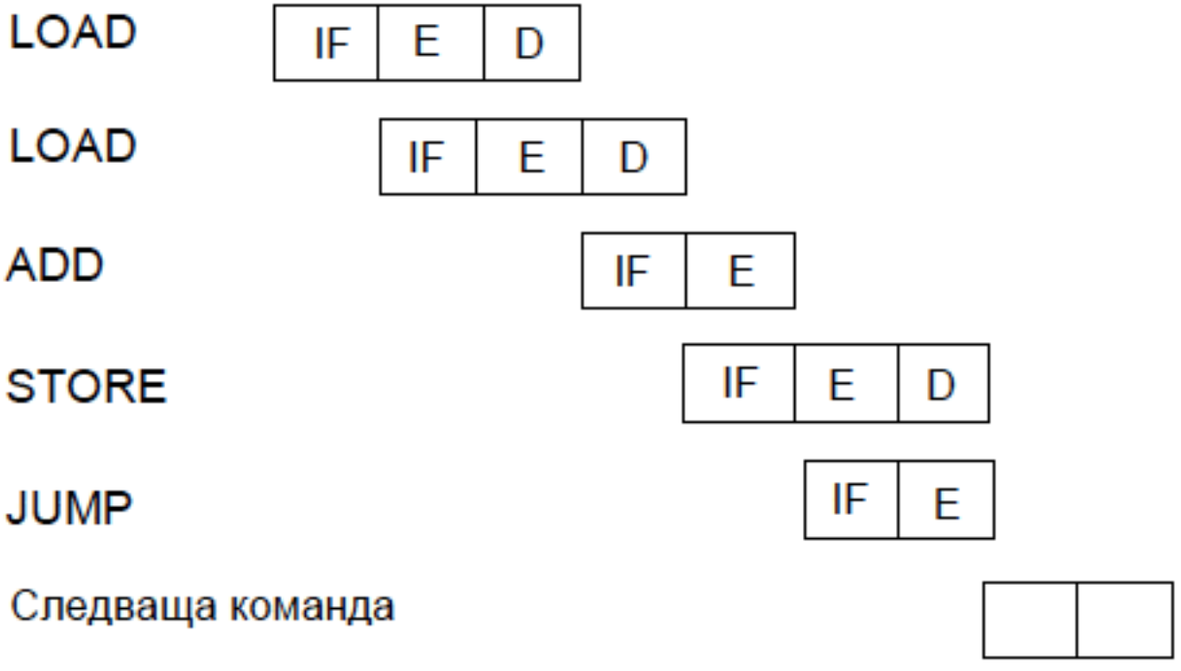
А): Поради наличието на обща памет за данни и команди в този случай фази **IF** и **D** от различните команди не могат да се стартират едновременно в един и същ такт, защото се прави опит за заемане на един и същ ресурс (паметта). Изпълнението в този случай на програмния фрагмент е показано на фиг.5-4.



Фиг.5-4.Конвейерно изпълнение на програмата при наличието на обща памет за данни и команди

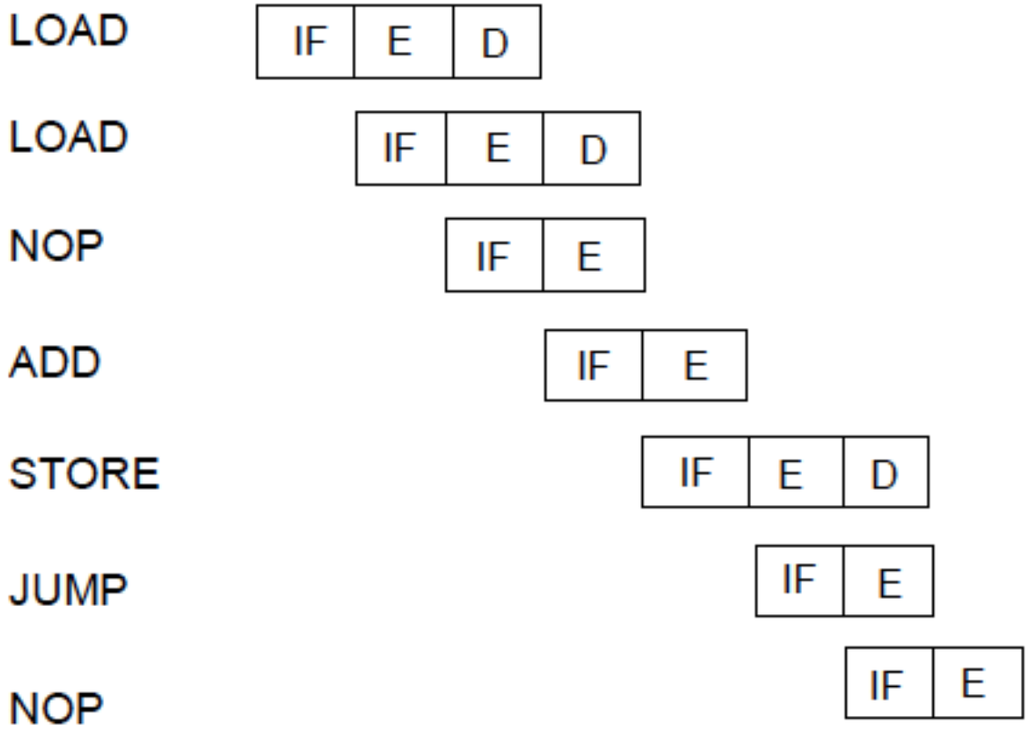
Б): Тук за разлика от предходния случай фази **IF** и **D** могат да стартират едновременно в един и същ такт, защото използват данни записани в две различни памети.

На фиг.4-9 е показано изпълнението на програмния фрагмент.

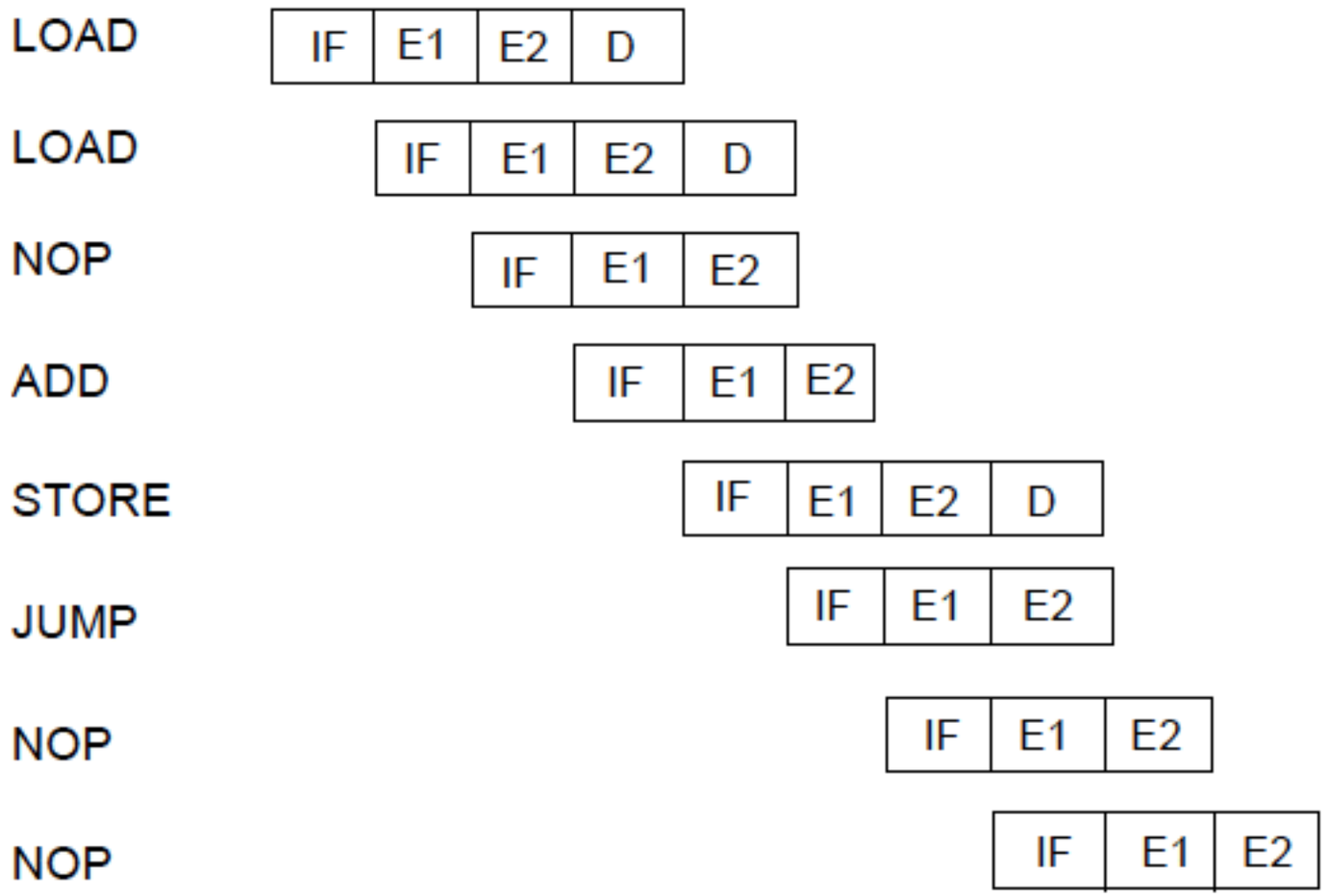


Фиг.4-9. Конвейерно изпълнение на програмата при наличието на отделни памети за данни и команди

В разгледаните по-горе ситуации, проблемите възникващи при конвейерното изпълнение на една програма, бяха решавани на апаратно ниво. Освен на апаратно ниво проблемите могат да бъдат разрешени и на програмно ниво. Това ще бъде демонстрирано само за процесор с отделни памети за данни и команди - фиг.4-9.



Фиг.4-9. Конвейерно изпълнение на програмата при наличието на отделни памети за данни и команди с използване на команда от типа NOP.



Фиг.4-10. Допълнително увеличаване на производителността чрез въвеждане на конвейер в изпълнителната степен

От изложеното в т.1. става ясно, че всяка команда за преход води до нарушаване на естествената последователност от постъпващи команди в конвейера. В резултат на което настъпват изчиствания на конвейера и последващо запълване с команди от новия клон на програмата. Така не е възможно да се получава резултат на всеки такт на конвейера. С увеличаване на броя степени на конвейера тези проблеми се усложняват. Ето защо при проектирането на конвейера се отделя специално внимание и се предприемат мерки за да се държи конвейера максимално натоварен при изпълнението на командите за преход. Някои от най-разпространените подходи за преодоляване на проблема са:

- предварителен избор на адреса на разклонение;
- използуване на няколко потока команди;
- прогнозирано разклонение;
- отложено разклонение.

Първите три подхода решават проблема на апаратно ниво, а последния – на програмно ниво. Последният се дискутира в точка 3.2.2 и затова тук ще се спрем само на апаратните възможности.

За да изложим по-ясно идеите за решаване на междукомандните зависимости, да въведем следните две определения [1]:

Определение 1. Област на определяне на командата i, означавана с D(i), това е множеството от всички обекти (регистри, клетки от паметта, флагове), съдържанието на които по един или друг начин влияе на изпълнението на команда i.

Определение 2. Област от значения на командата i, означавана с R(i), това е множеството от всички обекти (регистри, клетки от паметта, флагове), съдържанието на които може да бъде изменено в резултат на изпълнението на команда **i**.

По този начин D(i) е множеството на всички обекти четени от команда i, а R(i) е множеството от всички обекти модифицирани от нея. Следователно, взаимното влияние между команди i и j възникват когато е налице пресичане на двете области: R(i) и D(j), D(i) и R(j) или R(i) ∩ R(j).

5.**2.1.** На апаратно ниво

За откриване на смущенията се използват следните прийоми:

а) Обикновено в първата степен се определят областите D(i) и R(i) на командата i и се сравняват с тези на намиращите се в конвейера команди.

б) Командата се изпълнява в конвейера дотогава докато не се достигне точка, в която се изисква елемент или от D(i) или от R(i). Тогава се осъществява проверка за това има ли междукомандни зависимости с намиращите се в конвейера команди.

Вторият подход е по по-гъвкав, спиранията и изчистванията на конвейера са по-редки, но изисква и по-голям обем от апаратура.

За отстраняване на смущенията се използват следните прийоми.

а) Спира се работата на конвейера за i-та команда и за всички следващи след нея команди, докато командата, намираща се в конвейера и имаща конфликтна ситуация с i-та, не напусне конвейера.

б) Преустановява се изпълнението на команда i, но на команди i+1, i+2... се разрешава предвижването по конвейера. Това дава възможност да се задмине i-та команда. Естествено команди i+1, i+2...се предвижват по конвейера само ако нямат задръжки, както с намиращите се команди в конвейера, така и с команда i.

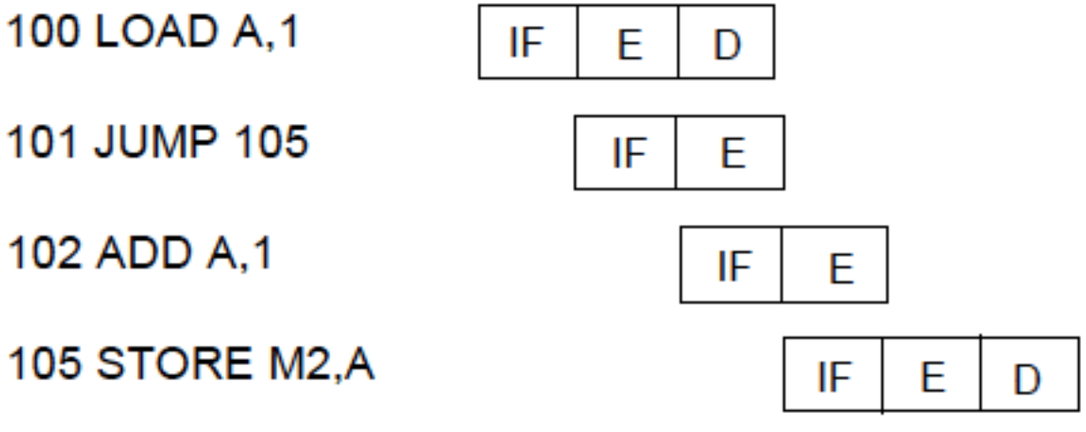
5.2.2. На програмно ниво

За отстраняване на споменатите зависимости между командите, се въвеждат команди от типа **NOP**, както беше посочено в т.3. Този способ се нарича отложено разклонение и спомага за увеличаване на ефективността на конвейера. Един друг способ е чрез реорганизация на програмния код и се нарича оптимизирано разклонение. Неговата същност е разгледана по-надолу, на базата на един програмен фрагмент. За сравнение е показано и отложеното разклонение.

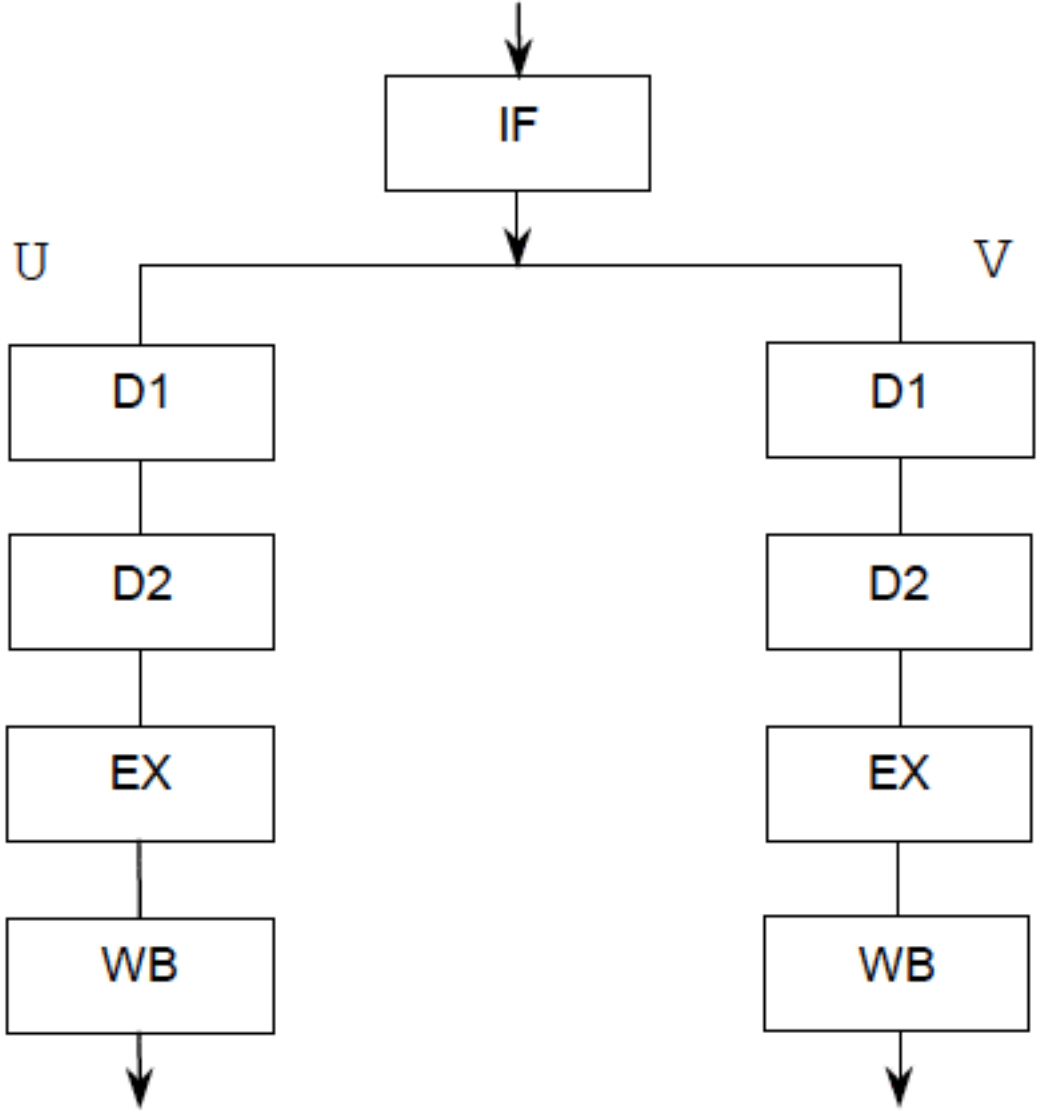
Оригиналният програмен фрагмент, изпълняван на не конвейерен процесор е показан в първа колона на таблицата (тук, както и по-надолу, числото пред командата е адрес на клетка от паметта); във втора колона е показана програмата при отложено разклонение; в трета колона е показана програмата при оптимизирано разклонение:

Оригинален код	Отложено разклонение	Оптимизирано разклонение
100 LOAD A,M1	100 LOAD A,M1	100 LOAD A,M1
101 ADD A,1	101 ADD A,1	101 JUMP 105
102 JUMP 105	102 JUMP 106	102 ADD A,1
103 ADD A,B	103 NOP	103 ADD A,B
104 SUB A,C	104 ADD A,B	104 SUB A,C
105 STORE M2,A	105 SUB A,C	105 STORE M2,A
	106 STORE M2,A	

При прилагане на техниката на оптимизираното разклонение се получава програма, чиято основната разлика между нея и оригиналната е в разменените места (клетки от паметта) на командите **ADD A,1** и **JUMP 105.** Независимо от тази размяна резултатът се получава коректен. Действително, докато се установява новото съдържание на програмния брояч, определено от командата **JUMP,** първата степен на конвейера извлича съдържанието на клетка **102** и докато се актуализира съдържанието на регистър **A,** коректно се извлича командата **STORE** - фиг.5-11.



Фиг.5-11. Изпълнение на програмата с прилагане на метода на оптимизираното разклонение.



Фиг.5-12. Обобщена структура на конвейерите

За всеки един от конвейерите **U** и **V**, изпълнението на командите протича на пет фази:

Instruction **F**etch (**IF**) – извличане на командата;
Instruction **D**ecoding (**D1**) – декодиране на командата;
Address **D**ecoding (**D2**) – декодиране на адреса;
Instruction **EX**ecution (**EX**) – изпълнение на командата (обръщение към АЛУ и кеш паметта за данни);
Write **B**ack (**WB**) –обратно записване (актуализиране на регистровия файл).

В първата фаза - **IF** командите се извличат от вградения кеш или от паметта. Тъй като процесорът има отделни кеш памети за код и за данни, няма условия за възникване на колизии между извличането на команди и достъпа до данните. Две независими двойки буфери за извличане на командите от по 32 бита дават възможност за препокриване на извличането и на изпълнението на кода на двата конвейера. Тези буфери се препокриват с буфера за цел на прехода (**B**ranch **T**arget **B**uffer - **BTB**), който се използва за предсказване на преходите (с цел избягването конвейерните прекъсвания, които могат да възникнат, ако необходимият код не се намира в конвейера). В даден момент е активен само един буфер за извличане, в който постъпва поредната команда от кеш паметта. Това е така, докато се извлече команда за преход. В такъв случай чрез BTB се предсказва дали преходът в програмата ще бъде изпълнен или не.
Ако се прецени, че преходът няма да се изпълни, продължава последователното извличане на команди. Ако се прецени, че преходът ще бъде изпълнен, командите започват да постъпват в другия буфер за извличане, сякаш той вече е изпълнен. Ако предвиждането се окаже погрешно, конвейерът се изчиства и извличането на кода се връща на първия буфер.

Във втората фаза - **D1** се определя типа на командата и на основа на следните правила се взема решение за разпаралелване на командите (дали да се разпределят на една, или две команди за двата целочислени конвейера):

- Двете следващи команди трябва да са "прости", т.е. да са изцяло апаратно реализируеми.
- Безусловните преходи JMP, условните преходи Jcc near и извикването на функции могат да бъдат сдвоени само тогава, ако се случи те да са втора команда в двойката Това означава зареждане във **V** конвейера.
- Преместването с една позиция (SAL/SAR/SHL/SHR reg/mrm,1) или непосредствен операнд (SAL/SAR/SHL/SHR reg/mrm,imm) или ротация на една позиция (RCL/RCR/ROL/ROR reg/mem,1), трябва да бъде първа команда в двойка команди - това означава, че те могат да бъдат зареждани само във **V** конвейер.
- Не трябва да съществува зависимост по регистри. Например, следната последователност:

```
mov ax,5
inc ax
```

не може да се разпаралели.

- Командите с префикс (с някои изключения) могат да се изпълняват само на конвейер **U**.
- В двете паралелно изпълнявани команди не трябва едновременно да има операнди за отместване и непосредствени операнди.

В третата фаза – **D2** се определят адресите на операндите. Командите, в които има непосредствени операнди и които освен това съдържат отмествания, както и команди, които използват едновременно базово и индексно адресиране, могат да бъдат изпълнявани на един такт.

В четвърта фаза – **EX** се извършват операции с АЛУ и обръщения към кеш паметта. Като правило, в тази фаза са необходими повече от един такт за изпълнението на командите. Тук се прави проверка на командите, намиращи се в конвейерите **U** и **V** по отношение на коректността на предсказване на преходите. Изключение представляват безусловните преходи, които се проверяват в следващата фаза.

В пета фаза – **WB** се записва резултатът в регистрите. Също така се прави запис и във флаговите регистри. Тук още веднъж се проверява коректността на условните преходи, защото по време на тяхното преминаване през отделните фази на конвейера може да са настъпили прекъсвания на конвейера.

Предсказване на преходите.

За да може да се предскаже дали даден преход ще бъде изпълнен или не в една програмата, процесорът използва динамичен алгоритъм за предсказване, работещ съвместно с **BTB** буфера. **BTB** буферът може да се разглежда като малка кеш памет, всеки запис на която се състои от програмен адрес на командата за преход и съответен краен адрес. Допълнително се използват битове за предисторията (History Bits), с който преходът се маркира съответно като "взет" или "не взет" като по този начин при повторно изпълнение програмния поток може да се оптимизира. Така се избягват прекъсвания на конвейера, които биха възникнали, ако се наложи да се извлече код от паметта. Както вече бе споменато, **BTB** е тясно свързан с буферите за извличане в първата конвейерна фаза. Ако при декодирането на командата във фаза **D1** на конвейера се открие програмно разклонение, програмният адрес на командата за преход се използва като обръщение в **BTB,** за да се определи целта на разклонението и да се стартира механизмът за извличане на код. При правилно предвиждане конвейерът не прекъсва. По този начин в един такт могат се изпълнят условни и безусловни преходи, както и да се извлекат NEAR процедури.

Разклонения в програмата могат да се изпълняват паралелно с други целочислени команди. При погрешно предвиждане на прехода следва пълно изчистване на целочислените конвейери и повторно извличане на програмния код.

Конвейерът за реални числа, или **FPU** (**F**loating **P**oint **U**nit) е основно преработен, но независимо от това, той запазва съвместимостта с аритметичния копроцесор 80387 и вграденото **FPU** на 486. Той представлява 8 степенен конвейер с отделни суматор, умножител и делител. **FPU** удовлетворява спецификациите на стандартите IEEE 754 и по-новия IEEE 854. Той е проектиран така, че на всеки процесорен такт се изпълнява по една операция с плаваща запетая. Ако втората команда е **FXCHG** (**F**loating point e**XCH**an**G**e), могат да се изпълняват и две операции за един такт.

Отделните фази на конвейера са:

IF – Извличане;

D1 – Декодиране на командата;

D2 – Формиране на адреса;

EX - Превръщане на външния формат на данните с плаваща запетая във вътрешен и запис на операндите в регистровия файл.

X1 – първа фаза на изпълнението на операциите с плаваща запетая;

X2 – втора фаза на изпълнението на операциите с плаваща запетая;

WF – закръгление и запис на резултата в регистровия файл;

ER – съобщение за грешка и актуализиране думата на състоянието.

Първите пет нива от **IF** до **WB** се делят с **U** конвейера.

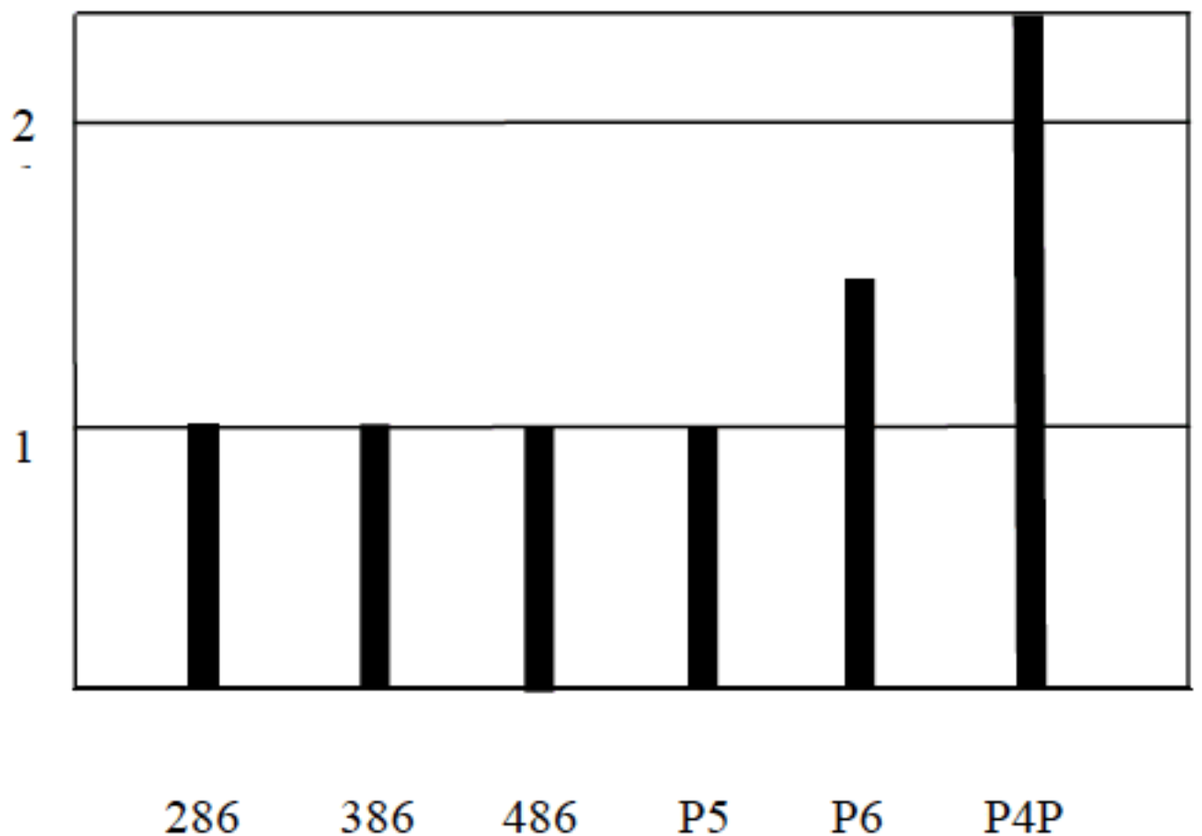
• Архитектурата на процесорите на Intel спада към класа процесори **CISC**, за които както знаем командите трудно се подлагат на конвейерно изпълнение. Ето защо всяка една x86 команда вътрешно се преобразува в подобни на **RISC** операции, наречени от Intel микрооперации (microops) Това спомага за опростяване на обработката на набора команди.

• В P6 се прилага стратегията за не поредно изпълнение, която позволява да се променя редът на изпълнение на командите. Като отделя настрана командите, които не могат да бъдат изпълнени веднага и обработва онези команди, които могат, P6 е способен да заобиколи някои от условията забавящи конвейерите на P5, чийто конвейери работят синхронно. Разбира се, взети са мерки за получаване на коректни резултати.

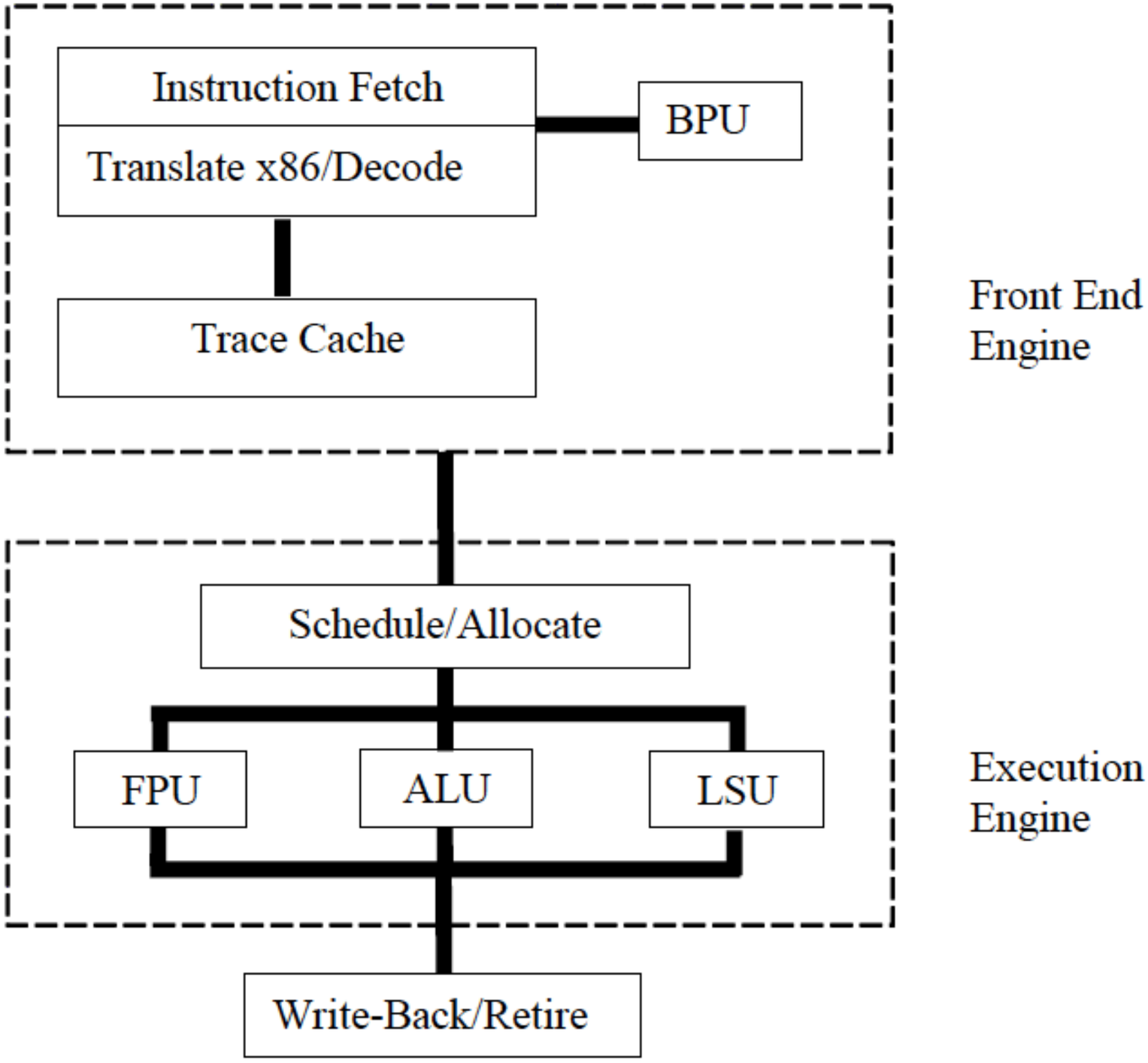
• Конвейерът на P6 е 14 степенен, разделен на три секции: поредно изпълнение, не поредно изпълнение, и оттегляне. Секцията за поредно изпълнение е разделена от секцията за непоредно изпълнение чрез микрооперационно буфериране в станцията за резервиране. Разделянето позволява трите секции да работят относително независимо, което подобрява взаимното съгласуване.

• И на края, P6 използва смяна на имената на регистрите, за да подпомогне непоредното изпълнение на командите и заобиколи друго класически тясно място в архитектурата си – ограничения брой регистри, дефинирани в набора регистри.

На фиг.5-13, предоставена от Intel, са показани последните 6 дизайна на Intel. По вертикалната ос е нанесена относителната честота на работа на процесорите, а по хоризонталната ос са показани различните процесори.



Фиг.5-13. Относителна честота на процесорите на Intel



Фиг.5-14. Обобщена структура на конвейерите в P4P.
(Съкращенията имат следния смисъл: BPU - Branch Prediction Unit; FPU - Floating Point Unit; LSU - Load/Store Unit)

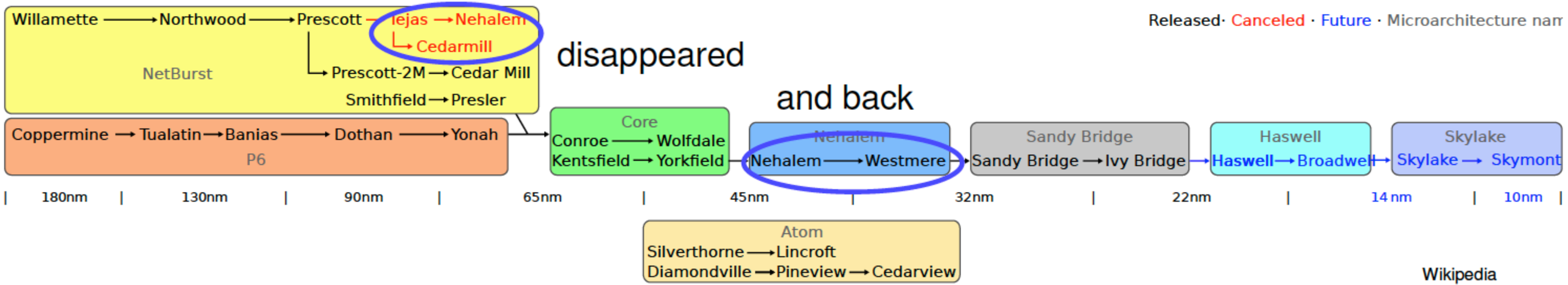
Кеш паметта за следите оперира в два режима:

- Режим на изпълнение;
- Режим на изграждане на следите;

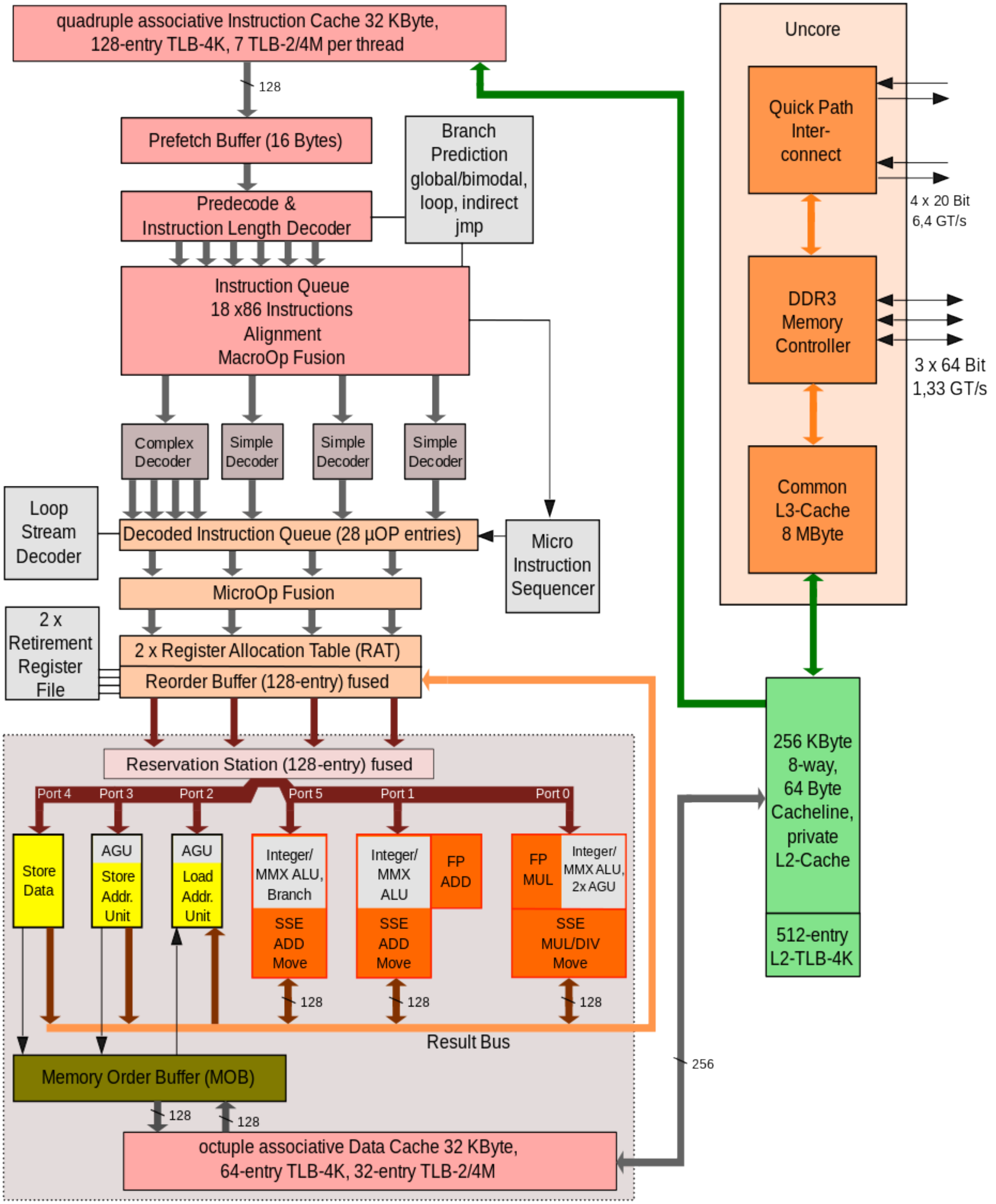
В първият режим кеш паметта за следите захранва със съхранените следи изпълнителната логика. Това е нормалният режим на работа. Когато нужните микрооперации отсъстват в L1, тогава се преминава към втория режим. В този режим машината за предварителна обработка извлича команди x86 от L2 кеш паметта, транслира ги в микрооперации, построява "сегмента на следите" с тях и зарежда сегмента в кеш паметта за следите за изпълнение. Кеш паметта за следите работи съвместно с **BPU** (**B**ranch **P**rediction **U**nit). Това е така, защото сегмента за следите е много повече отколкото само предварително извлечен код x86, транслиран и декодиран. Кеш паметта за следите използва предсказване на преходите като той е вграден в следата.

	стъпала	
Nehalem (Intel)	14	CISC, шестпътна суперскаларност, двупътна многонишковост, разбъркано изпълнение, интегриран контролер на паметта, до 24 MB кеш-памет, Turbo Boost, до 8 ядра
Bulldozer (AMD)	20	CISC, суперскаларен, разбъркано изпълнение, интегриран контролер на паметта, до 16 MB кеш-памет, виртуализация, Turbo Core, FlexFPU, до 16 ядра
MIPS	5	RISC
Alpha 21264	7	RISC, суперскаларен, четирипътна многонишковост, разбъркано изпълнение, интегриран контролер на паметта
UltraSPARC T1	6	RISC, дизайн по GNU GPL, многонишковост, до 8 ядра, 4 нишки на ядро, интегриран контролер на паметта
Cortex-A57	18	RISC, трипътна суперскаларност, разбъркано изпълнение
Loongson GS464E	9	RISC, на база MIPS, четирипътна суперскаларност, до 4 ядра, интегриран контролер на паметта

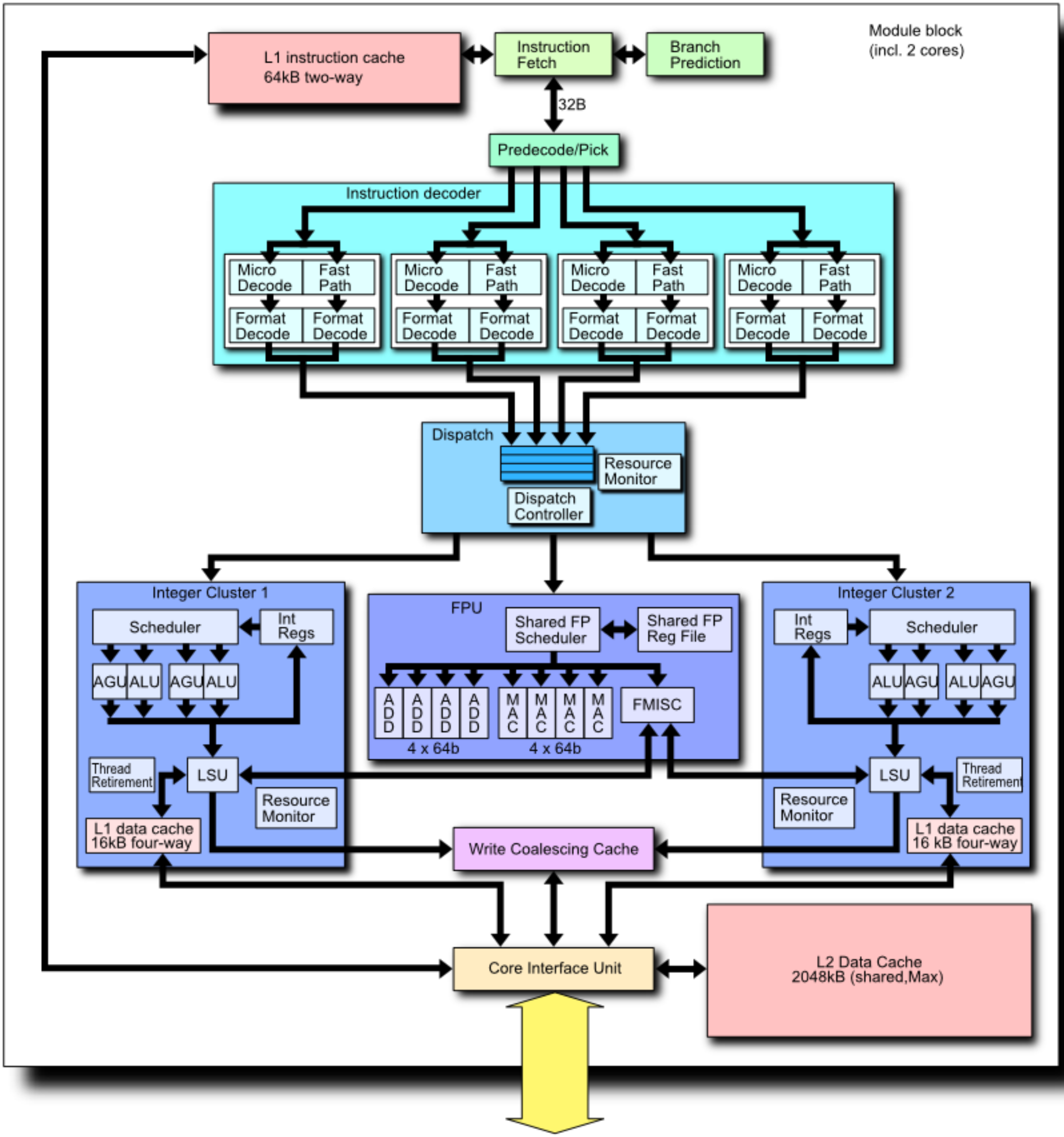
Intel Roadmap

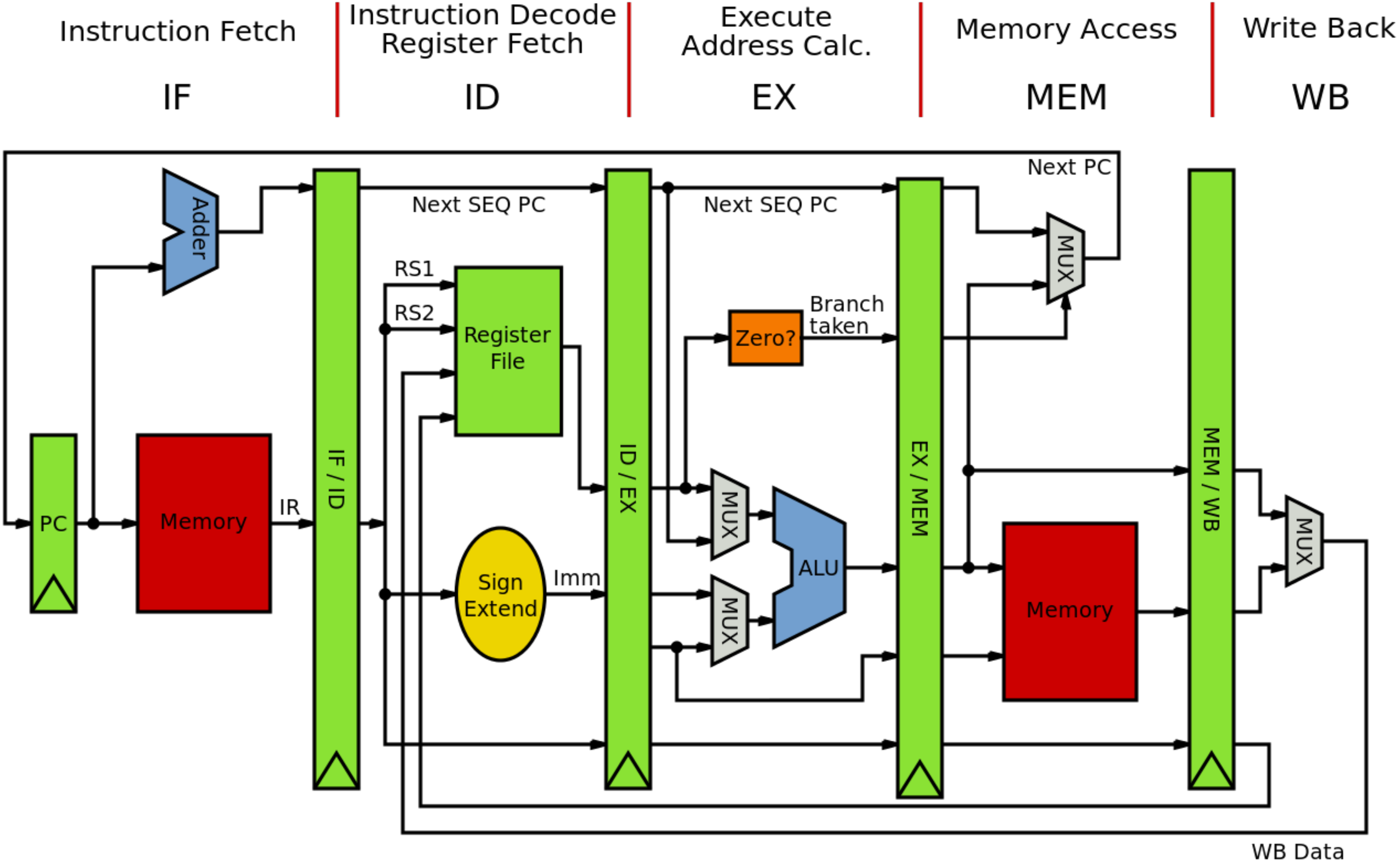


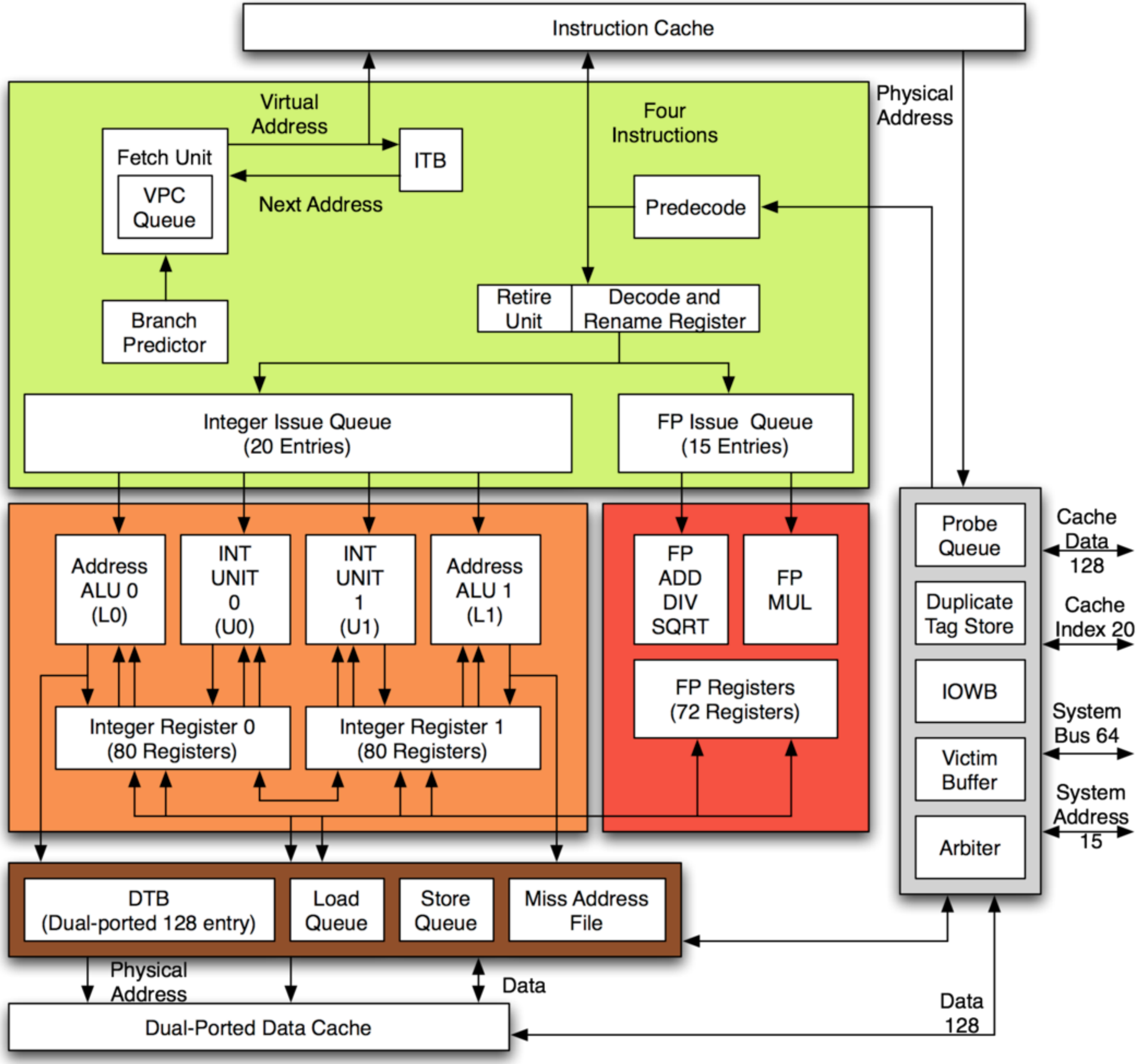
Intel Nehalem microarchitecture

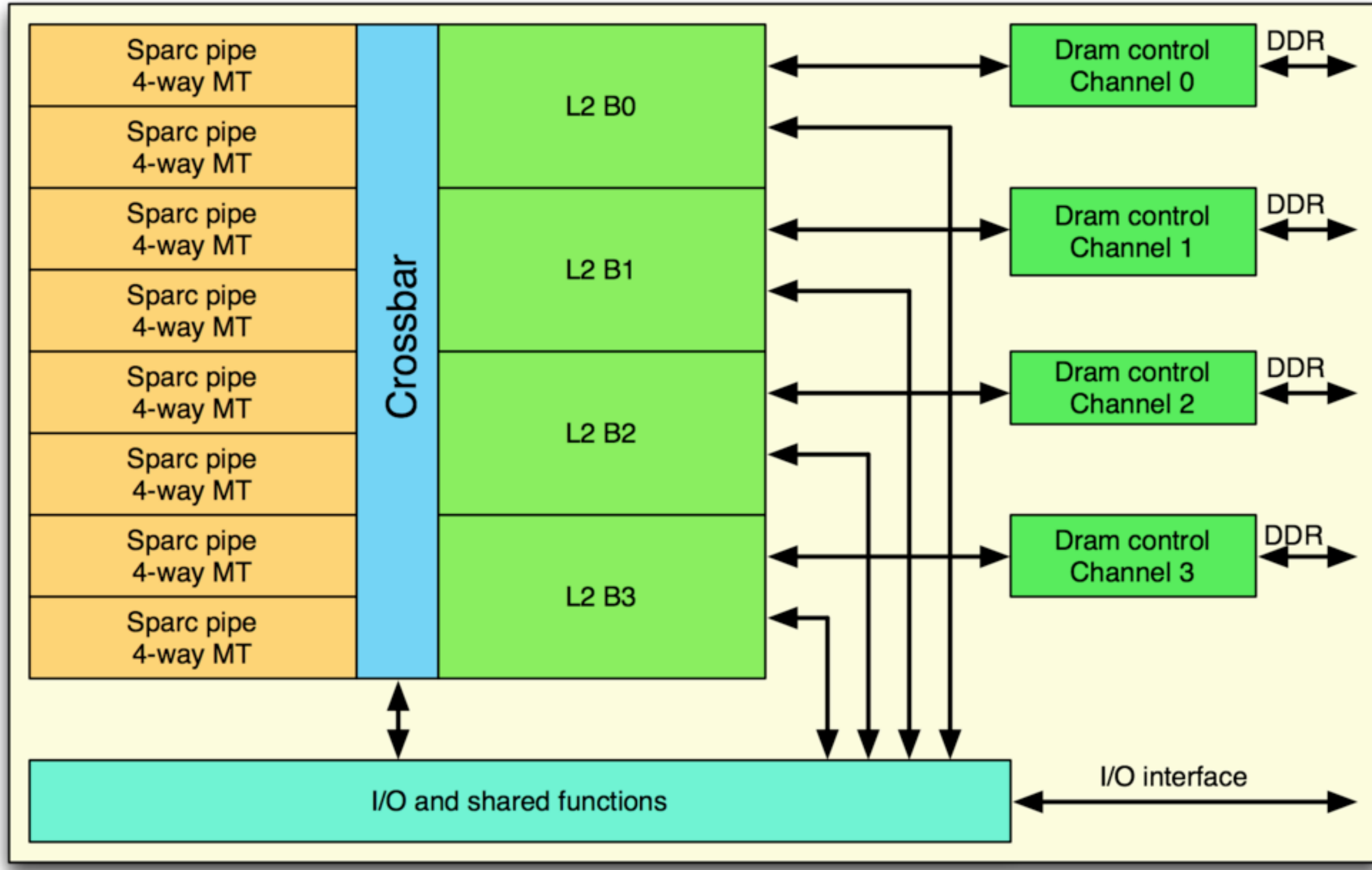


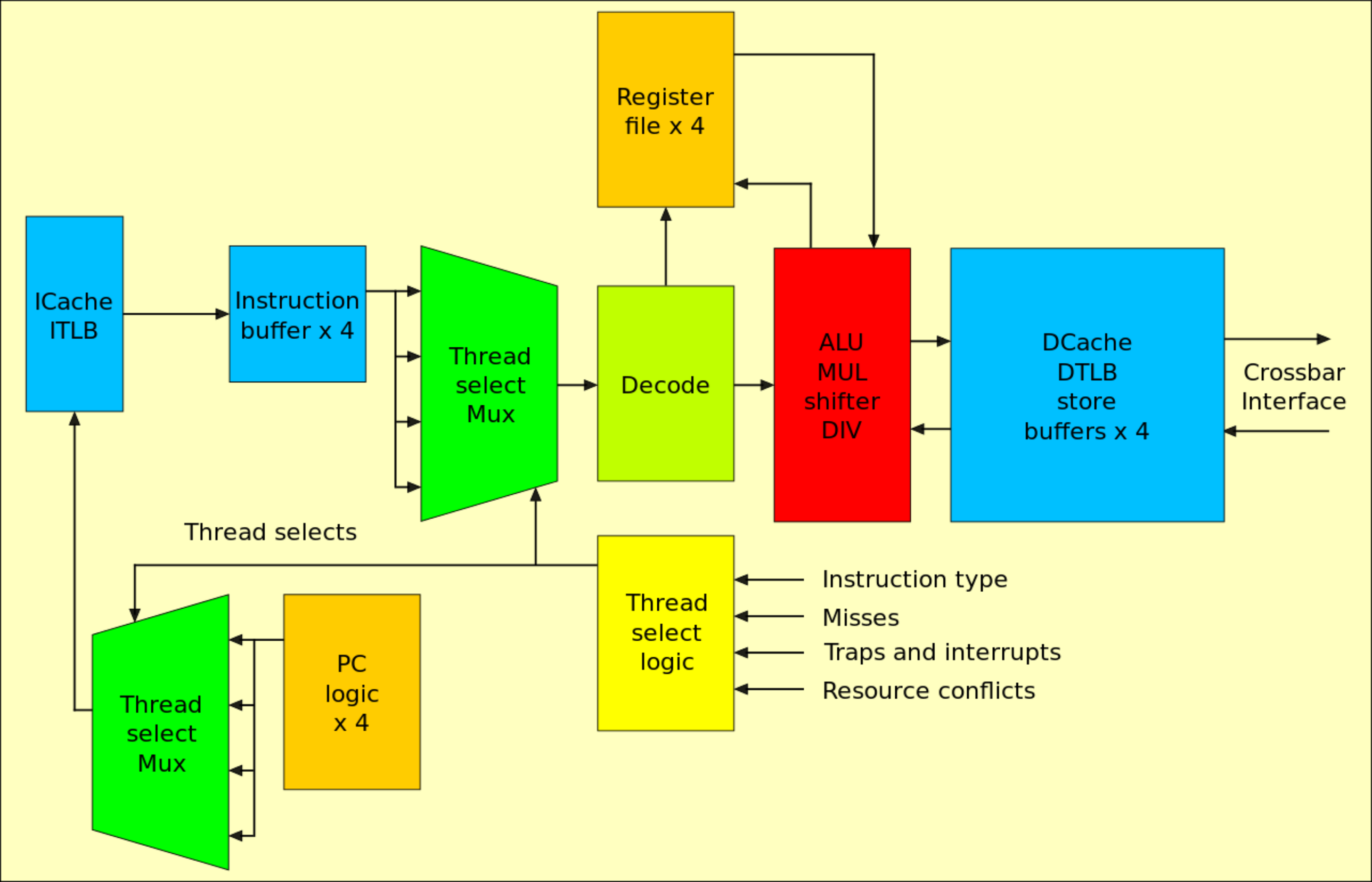
GT/s: gigatransfers per second


Module block
(incl. 2 cores)
L1 instruction cache
64kB two-way
Instruction
Fetch
Branch
Prediction
32B
Predecode/Pick
Instruction decoder
Micro
Decode
Fast
Path
Format
Decode
Dispatch
Resource
Monitor
Dispatch
Controller
Integer Cluster 1
Scheduler
Int
Regs
AGU
ALU
Thread
Retirement
LSU
Resource
Monitor
L1 data cache
16kB four-way
FPU
Shared FP
Scheduler
Shared FP
Reg File
ADD
MAC
FMISC
4 x 64b
Integer Cluster 2
Write Coalescing Cache
L1 data cache
16 kB four-way
Core Interface Unit
L2 Data Cache
2048kB (shared,Max)

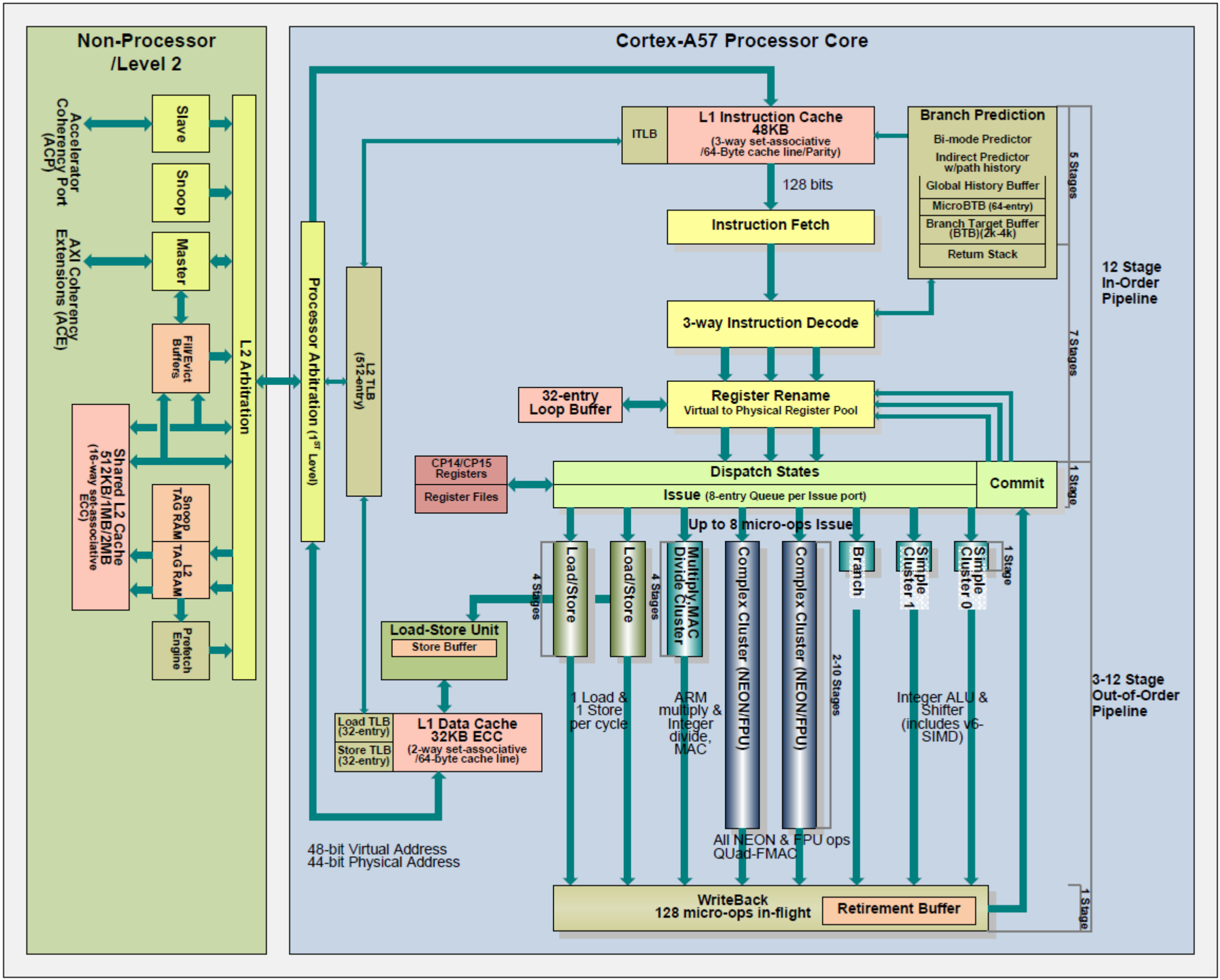

Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
MUX
RS1
RS2
Register
File
Zero?
Branch
taken
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
MUX
MUX
ALU
MUX
Sign
Extend
Imm
Memory
WB Data

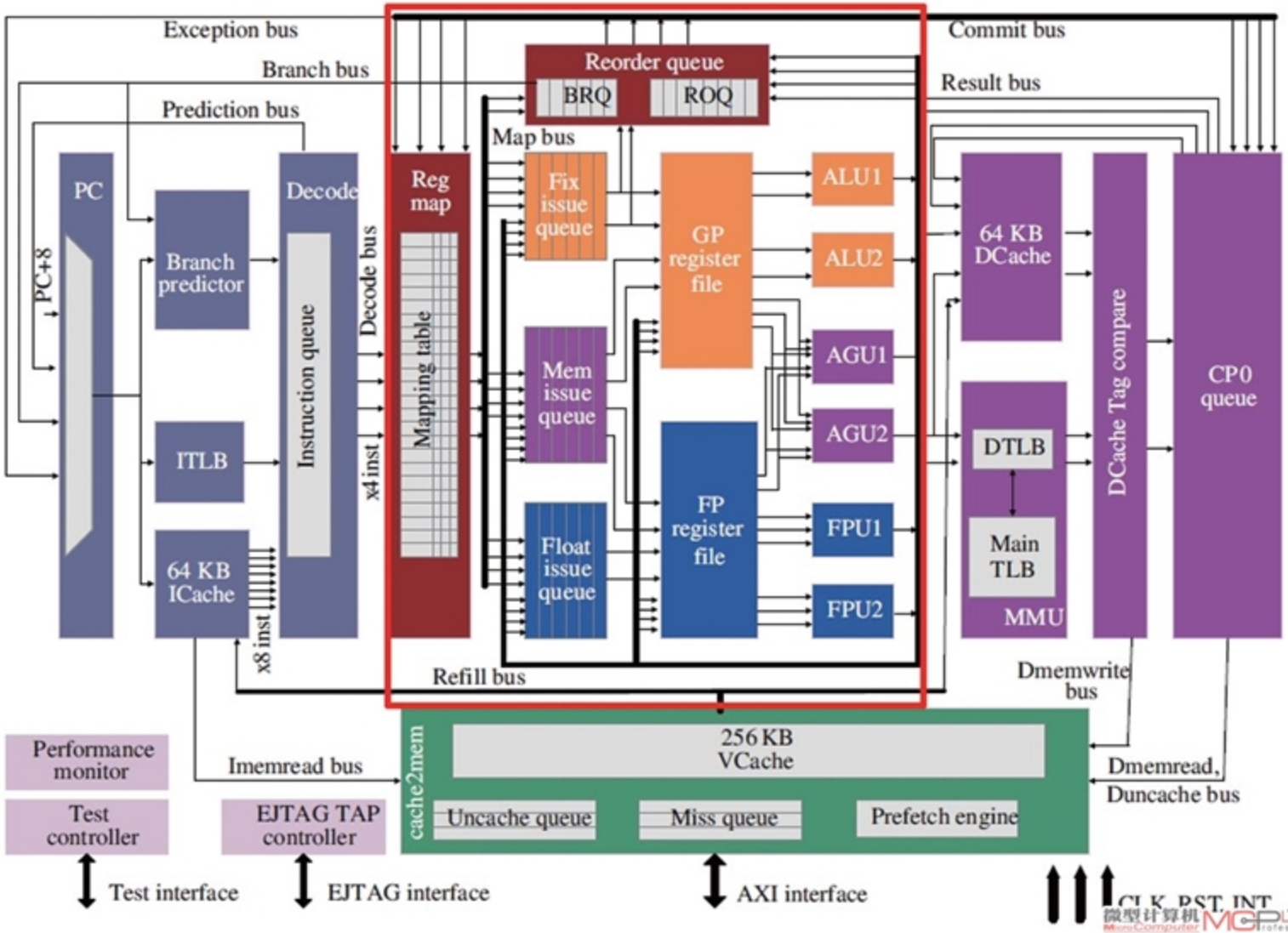

Instruction Cache
Virtual Address
Fetch Unit
VPC Queue
ITB
Next Address
Branch Predictor
Four Instructions
Predecode
Physical Address
Retire Unit
Decode and Rename Register
Integer Issue Queue (20 Entries)
FP Issue Queue (15 Entries)
Address ALU 0 (L0)
INT UNIT 0 (U0)
INT UNIT 1 (U1)
Address ALU 1 (L1)
Integer Register 0 (80 Registers)
Integer Register 1 (80 Registers)
FP ADD DIV SQRT
FP MUL
FP Registers (72 Registers)
Probe Queue
Duplicate Tag Store
IOWB
Victim Buffer
Arbiter
Cache Data 128
Cache Index 20
System Bus 64
System Address 15
DTB (Dual-ported 128 entry)
Load Queue
Store Queue
Miss Address File
Physical Address
Data
Dual-Ported Data Cache
Data 128

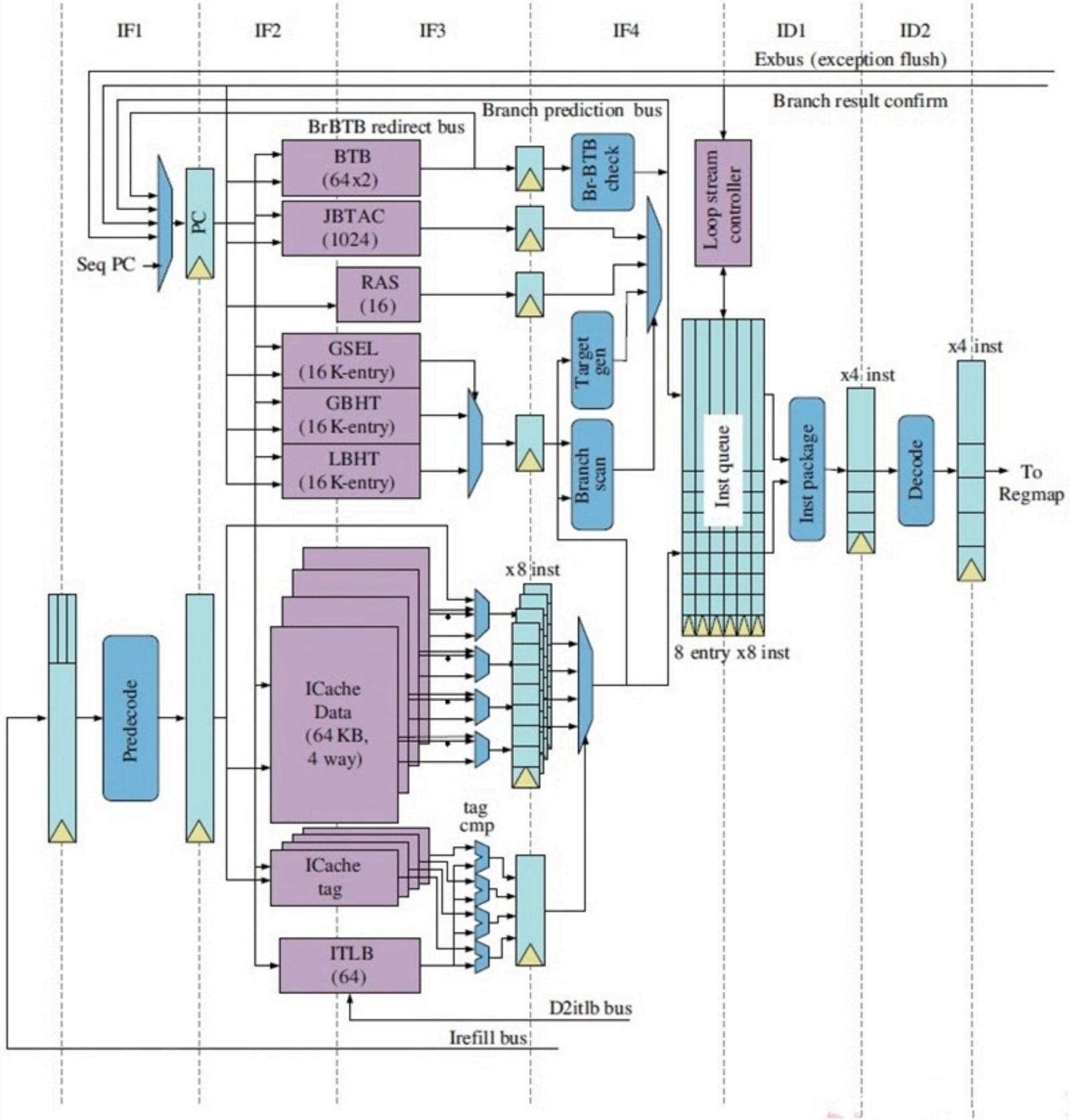

Sparc pipe
4-way MT
Sparc pipe
4-way MT
Sparc pipe
4-way MT
Sparc pipe
4-way MT
Sparc pipe
4-way MT
Sparc pipe
4-way MT
Sparc pipe
4-way MT
Sparc pipe
4-way MT
Crossbar
L2 B0
L2 B1
L2 B2
L2 B3
Dram control
Channel 0
Dram control
Channel 1
Dram control
Channel 2
Dram control
Channel 3
DDR
DDR
DDR
DDR
I/O and shared functions
I/O interface

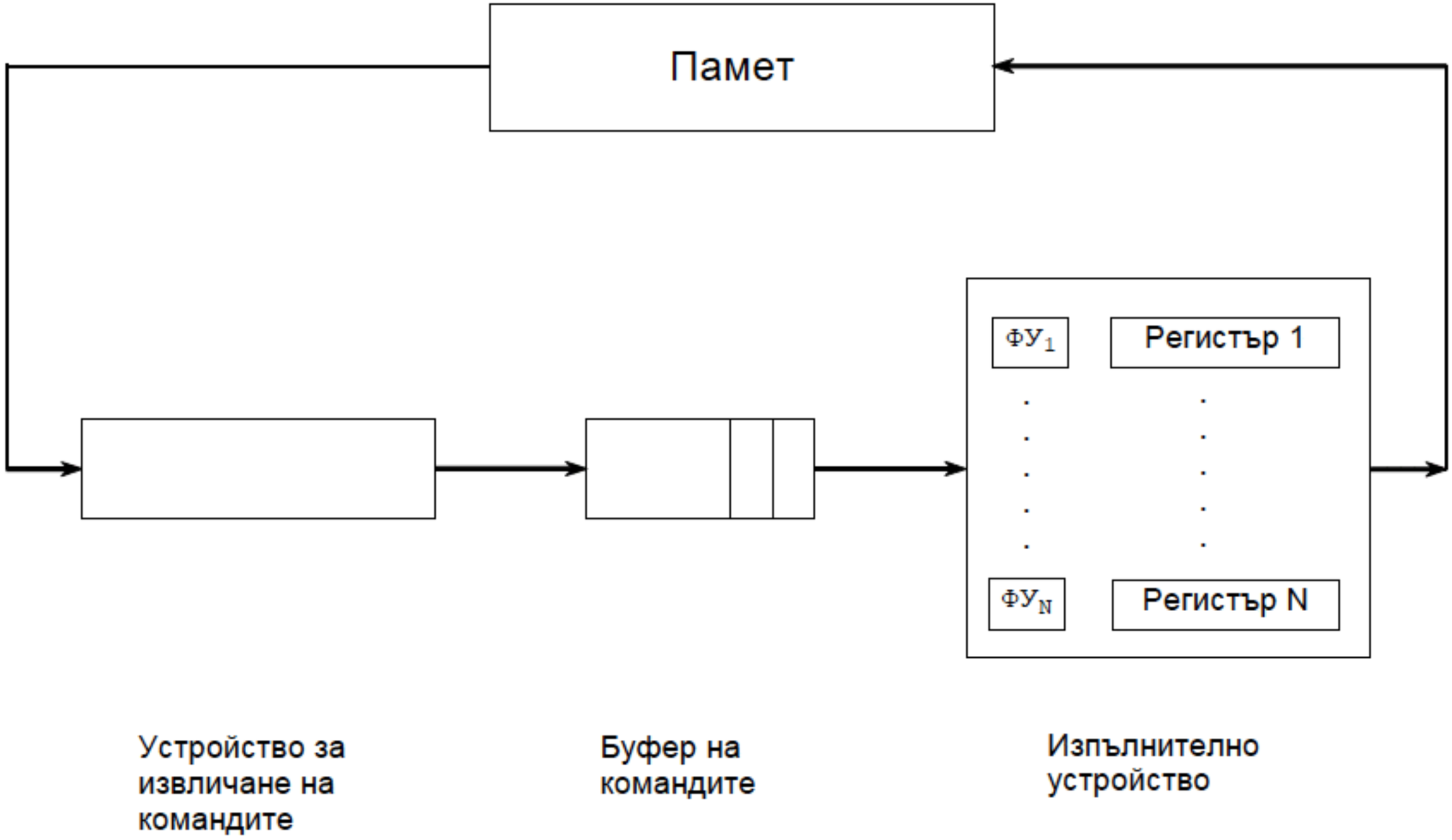

Register file x 4
ICache ITLB
Instruction buffer x 4
Thread select Mux
Decode
ALU MUL shifter DIV
DCache DTLB store buffers x 4
Crossbar Interface
Thread selects
Thread select logic
Instruction type
Misses
Traps and interrupts
Resource conflicts
Thread select Mux
PC logic x 4

ARM Cortex-A57 Block Diagram




Exception bus
Branch bus
Prediction bus
Commit bus
Result bus
Reorder queue
BRQ
ROQ
Map bus
PC
PC+8
Branch predictor
ITLB
64 KB ICache
Decode
Instruction queue
Decode bus
x4 inst
x8 inst
Reg map
Mapping table
Fix issue queue
Mem issue queue
Float issue queue
GP register file
FP register file
ALU1
ALU2
AGU1
AGU2
FPU1
FPU2
64 KB DCache
DTLB
Main TLB
MMU
DCache Tag compare
CP0 queue
Refill bus
Dmemwrite bus
Performance monitor
Test controller
Imemread bus
EJTAG TAP controller
cache2mem
256 KB VCache
Uncache queue
Miss queue
Prefetch engine
Dmemread, Duncache bus
Test interface
EJTAG interface
AXI interface
CLK, RST, INT


IF1
IF2
IF3
IF4
ID1
ID2
Exbus (exception flush)
Branch result confirm
Branch prediction bus
BrBTB redirect bus
BTB (64x2)
JBTAC (1024)
RAS (16)
GSEL (16 K-entry)
GBHT (16 K-entry)
LBHT (16 K-entry)
Br-BTB check
Target gen
Branch scan
Loop stream controller
Inst queue
Inst package
x4 inst
Decode
To Regmap
PC
Seq PC
Predecode
ICache Data (64 KB, 4 way)
x8 inst
8 entry x8 inst
tag cmp
ICache tag
ITLB (64)
D2itlb bus
Irefill bus



Фиг.5-1. Обобщена структура на процесор, реализиращ функционален паралелизъм

Разбира се, паралелното изпълнение на командите в суперскаларния процесор не винаги е възможно. Това може да е следствие на три причини.

• Конфликти при достъп до ресурси. Възникват, ако няколко команди едновременно се обърнат към един и същ ресурс напр. ФУ. Намаляване на отрицателния ефект от подобни ситуации се търси чрез дублиране на устройствата.

• Зависимост по управление. Тука има два аспекта. Първият – това е проблем, свързан с обработката на разклонения. Вторият е свързан с използването на команди с променлива дължина. В този случай избор на следващата команда не трябва да се прави, докато не завърши декодирането на предидущата команда. Ето защо суперскаларната архитектура е по-подходяща за **RISC** процесорите с техния фиксиран формат.

• Конфликти по данни. Причината за конфликти по данни се явяват зависимостите по данни между командите, когато следващата операция не може да бъде изпълнена ако й трябва резултата от предидущата операция. Тази зависимост е свойство на програмата и не може да бъде изключена по никакъв начин от компилаторът или с помощта на други апаратни средства. За избягването на престоите и образуването на мехурчета в конвейера, може да се натоварят устройствата с изпълнението на други команди, докато се формира резултата от предидущата операция, т.е. да се приложи стратегията за не поредно изпълнение на командите.

В суперскаларните процесори се използва динамично разпределение на командите, при което порядъка на тяхното избиране може да не съвпада с порядъка следван в програмата, но при това резултатът, естествено, трябва да съвпада с последователното изпълнение. За ефективна реализация на дадения подход, последователността от команди, от която се извършва избора, трябва да бъде голяма – необходимо е достатъчно голям прозорец на изпълнение (Window of Execution).

Прозорецът на изпълнение – това е набор от команди, които се явяват кандидати за изпълнение в даден момент. Всяка команда от този прозорец може да бъде изпълнена с отчитане на гореспоменатите ограничения. Количеството команди зависи от броя на конвейерите, включени в състава на процесора и броя на степените в конвейерите.

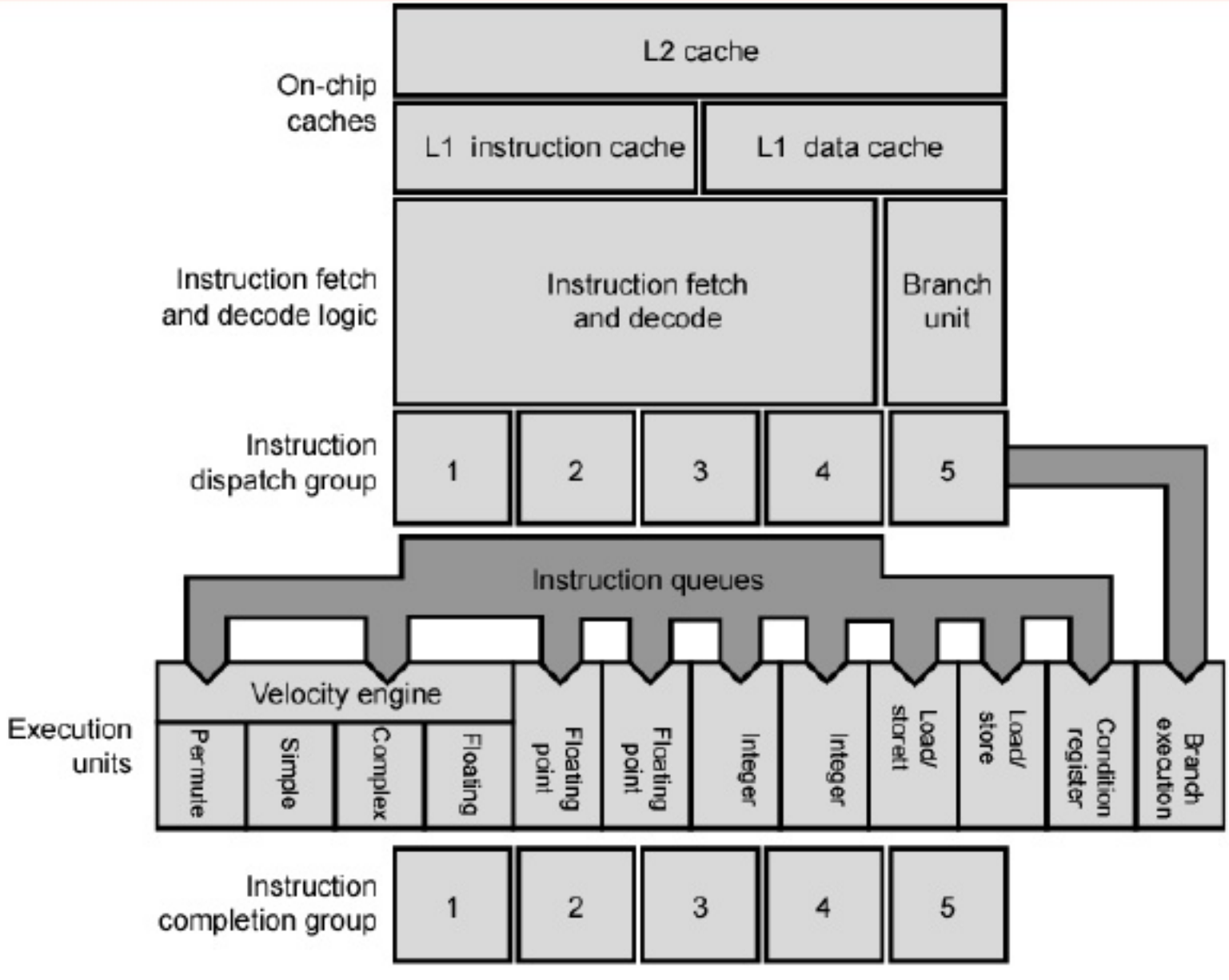
Показател за ефективната работа на конвейера се явява средния брой тактове за изпълнение на команда (**CPI** - **C**ycles **P**er **I**nstruction). Колкото тази величина е по-ниска, толкова е по-висока производителността на процесора. Идеалната величина е 1 за процесор с един единствен конвейер. При процесор с множество ФУ тази величина може да бъде и по-малка от единица, разбира се ако в един такт се изпълняват повече команди.

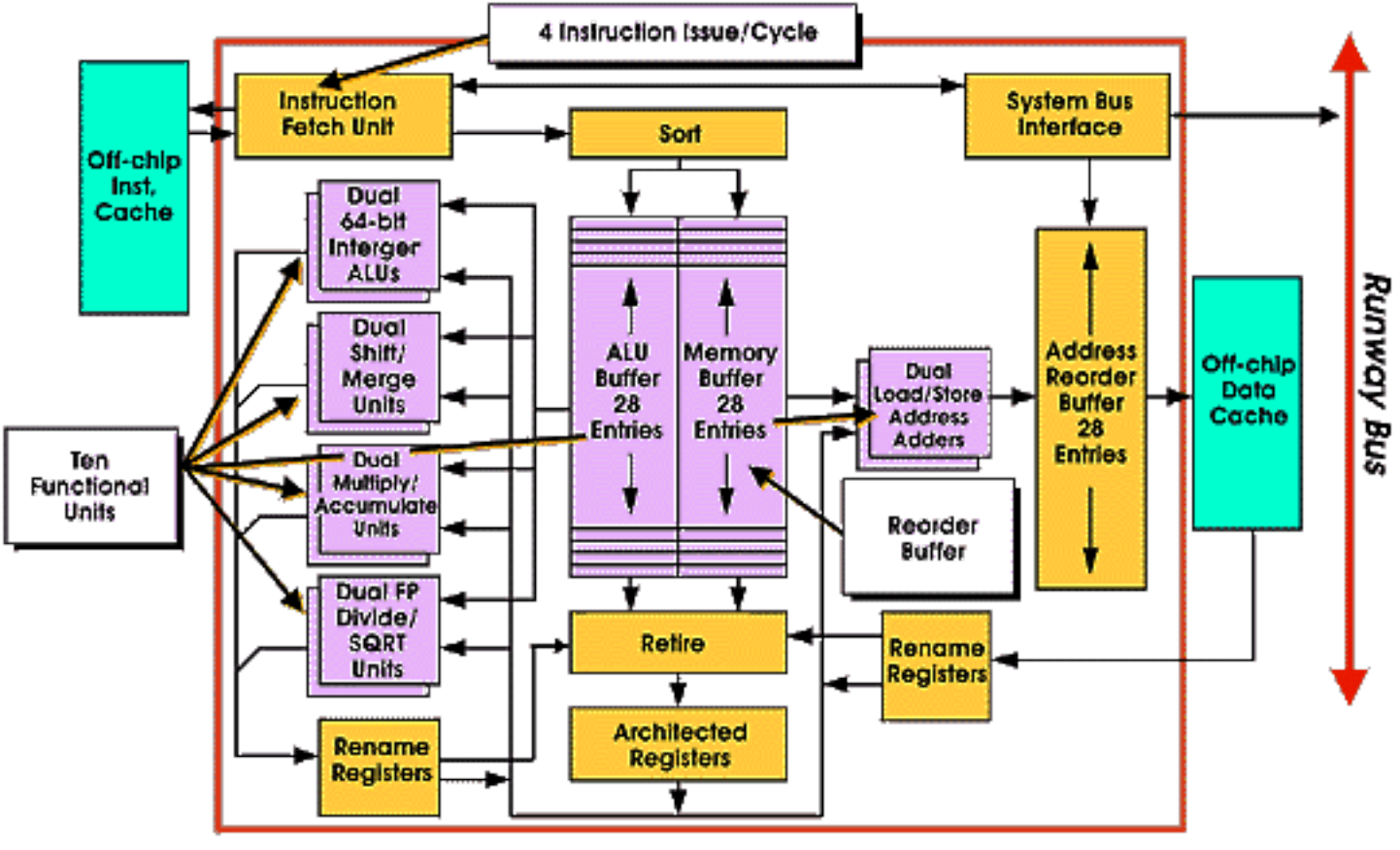
Понастоящем има две стратегии за постигане на по-висока производителност на процесор с ФУ, т.е. изпълнение на повече команди за секунда. Първата е "широк и плитък" конвейер, а втората е "тесен и дълбок" конвейер. Първата стратегия, прилага се напр. в процесора G4 на Motorola, използва повече на брой конвейери, но всеки от тях има малък на брой стъпала. Докато втората стратегия изисква по-малък брой конвейери, но за сметка на това те са с повече стъпала - напр. P4P на Intel.

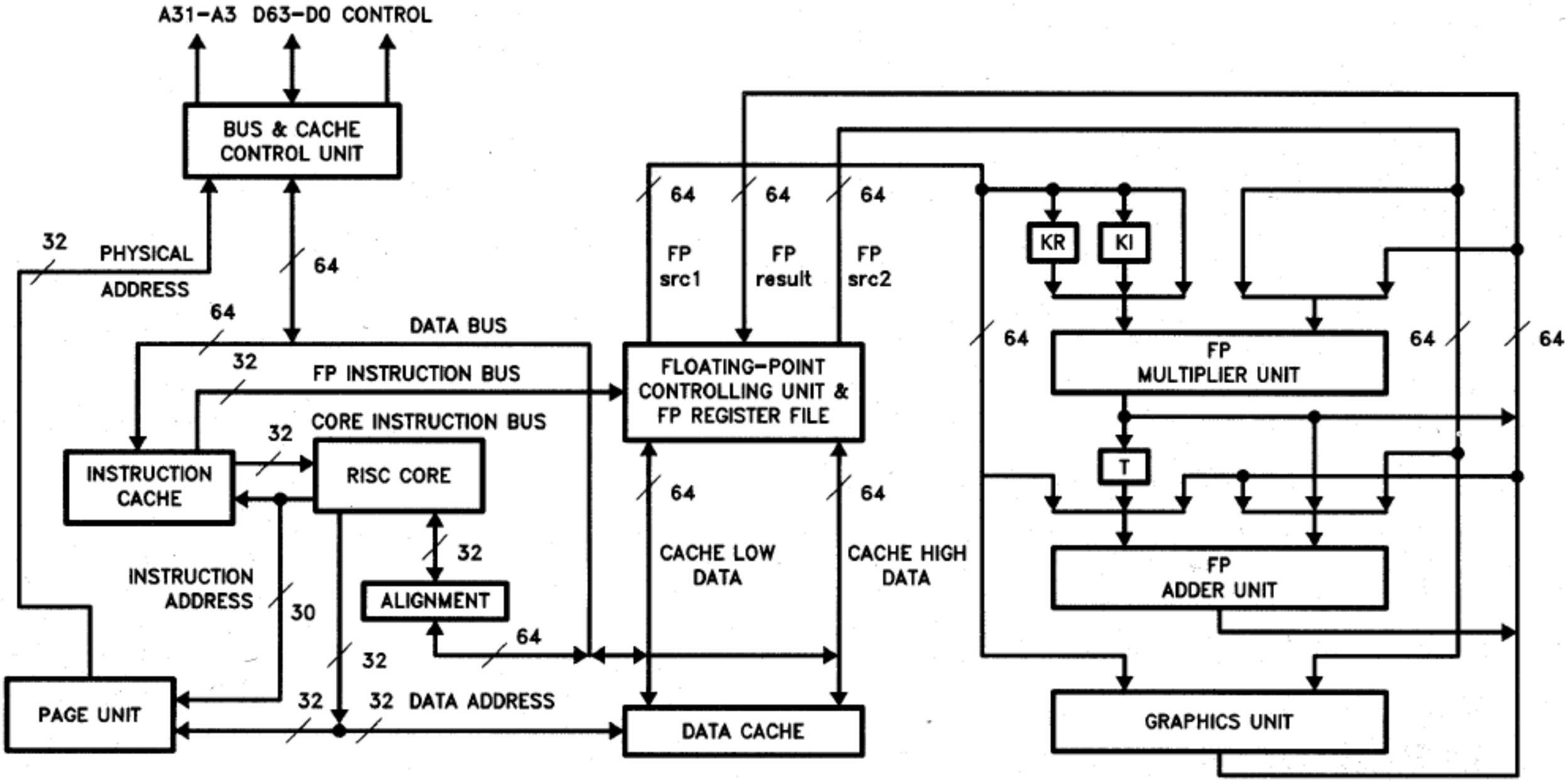
На следващите схеми:

- PowerPC 970 (G5)
- PA-RISC
- i860
- Itanium 2
- Efficeon
- Елбрус

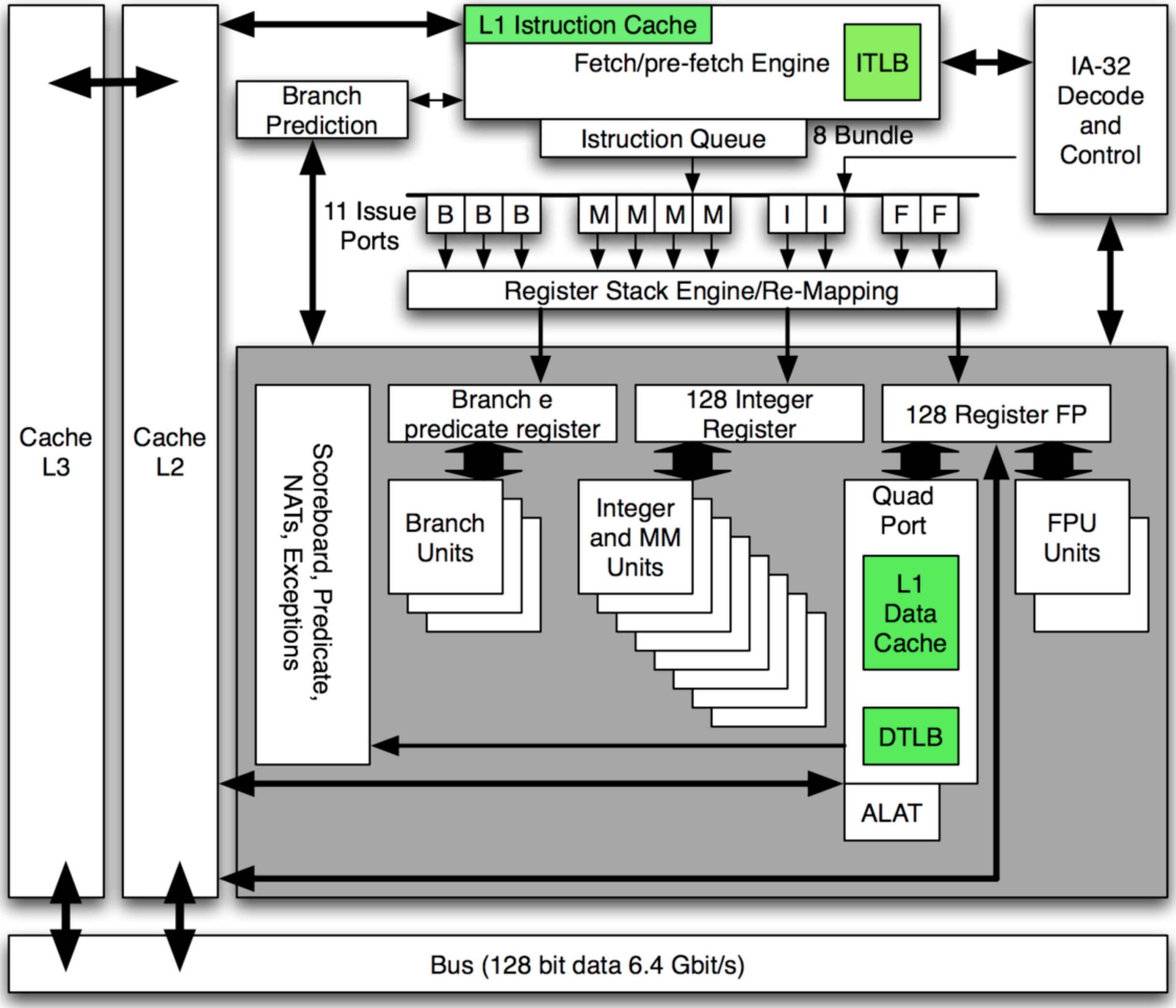
PowerPC G5 Block Diagram



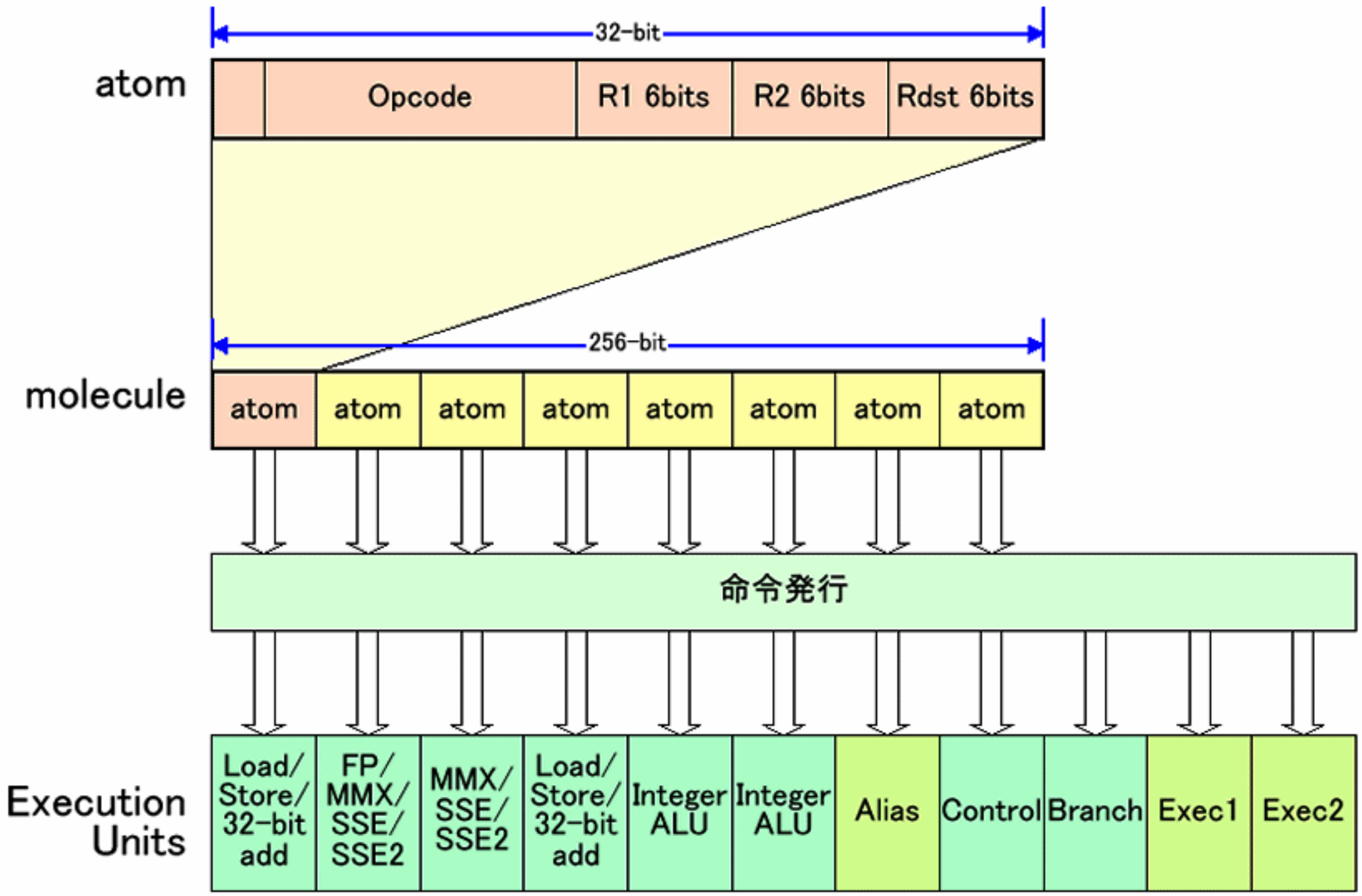

4 Instruction Issue/Cycle
Off-chip Inst. Cache
Instruction Fetch Unit
Sort
System Bus Interface
Dual 64-bit Interger ALUs
Dual Shift/ Merge Units
Dual Multiply/ Accumulate Units
Dual FP Divide/ SQRT Units
Ten Functional Units
ALU Buffer 28 Entries
Memory Buffer 28 Entries
Dual Load/Store Address Adders
Reorder Buffer
Address Reorder Buffer 28 Entries
Off-chip Data Cache
Runway Bus
Retire
Rename Registers
Architected Registers
Rename Registers

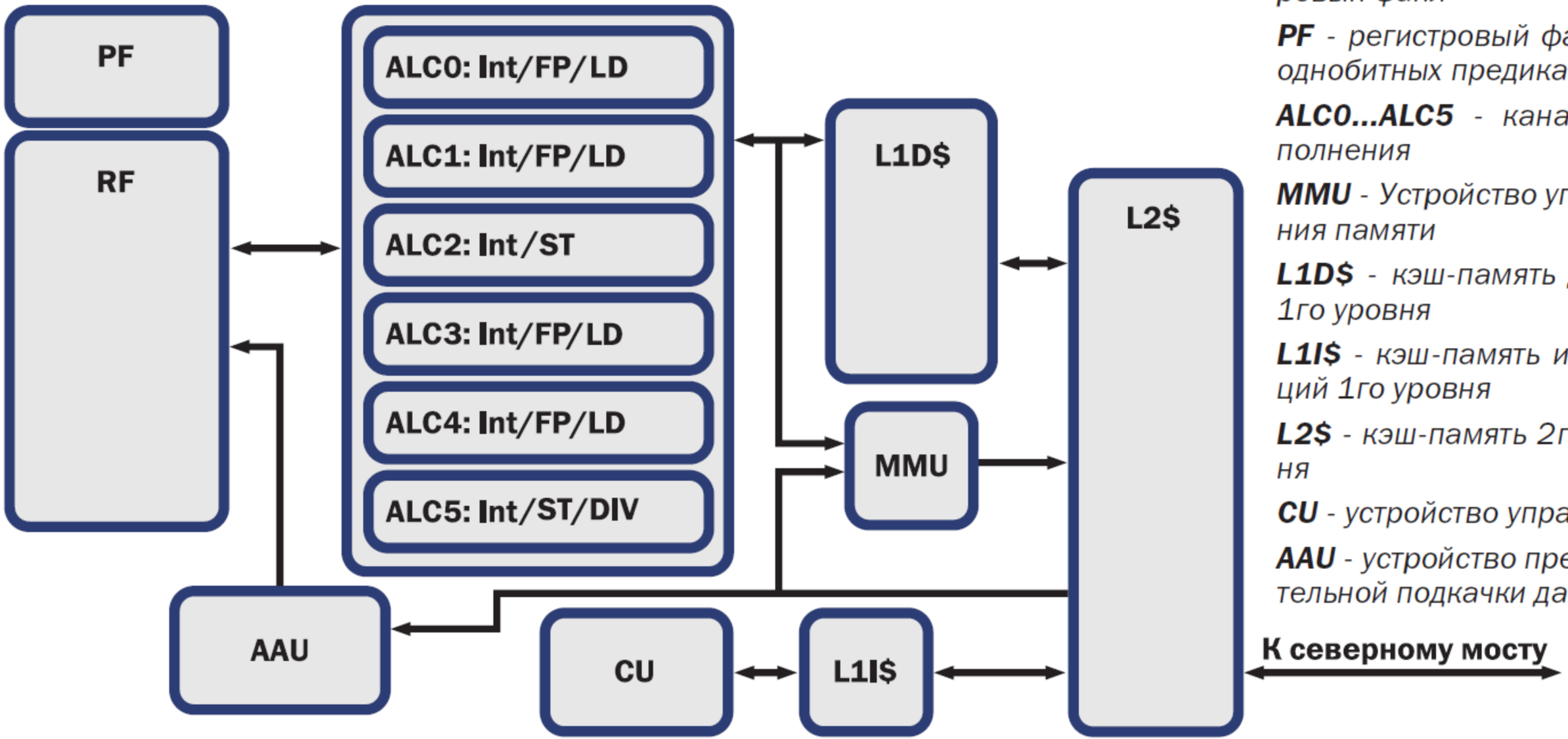

A31–A3
D63–D0
CONTROL
BUS & CACHE CONTROL UNIT
32
PHYSICAL ADDRESS
64
64
DATA BUS
32
FP INSTRUCTION BUS
32
CORE INSTRUCTION BUS
INSTRUCTION CACHE
RISC CORE
32
INSTRUCTION ADDRESS
30
ALIGNMENT
64
32
PAGE UNIT
32
32
DATA ADDRESS
64
64
64
FP src1
FP result
FP src2
FLOATING-POINT CONTROLLING UNIT & FP REGISTER FILE
64
64
CACHE LOW DATA
CACHE HIGH DATA
DATA CACHE
KR
KI
64
FP MULTIPLIER UNIT
64
64
T
FP ADDER UNIT
GRAPHICS UNIT

240296–1


L1 Istruction Cache
Fetch/pre-fetch Engine
ITLB
IA-32 Decode and Control
Branch Prediction
Istruction Queue
8 Bundle
11 Issue Ports
B B B
M M M M
I I
F F
Register Stack Engine/Re-Mapping
Cache L3
Cache L2
Scoreboard, Predicate, NATs, Exceptions
Branch e predicate register
128 Integer Register
128 Register FP
Branch Units
Integer and MM Units
Quad Port
L1 Data Cache
DTLB
ALAT
FPU Units
Bus (128 bit data 6.4 Gbit/s)

EfficeonのVLIW命令





RF *- универсальный регистровый файл*

PF *- регистровый файл для однобитных предикатов*

ALC0...ALC5 *- каналы исполнения*

MMU *- Устройство управления памяти*

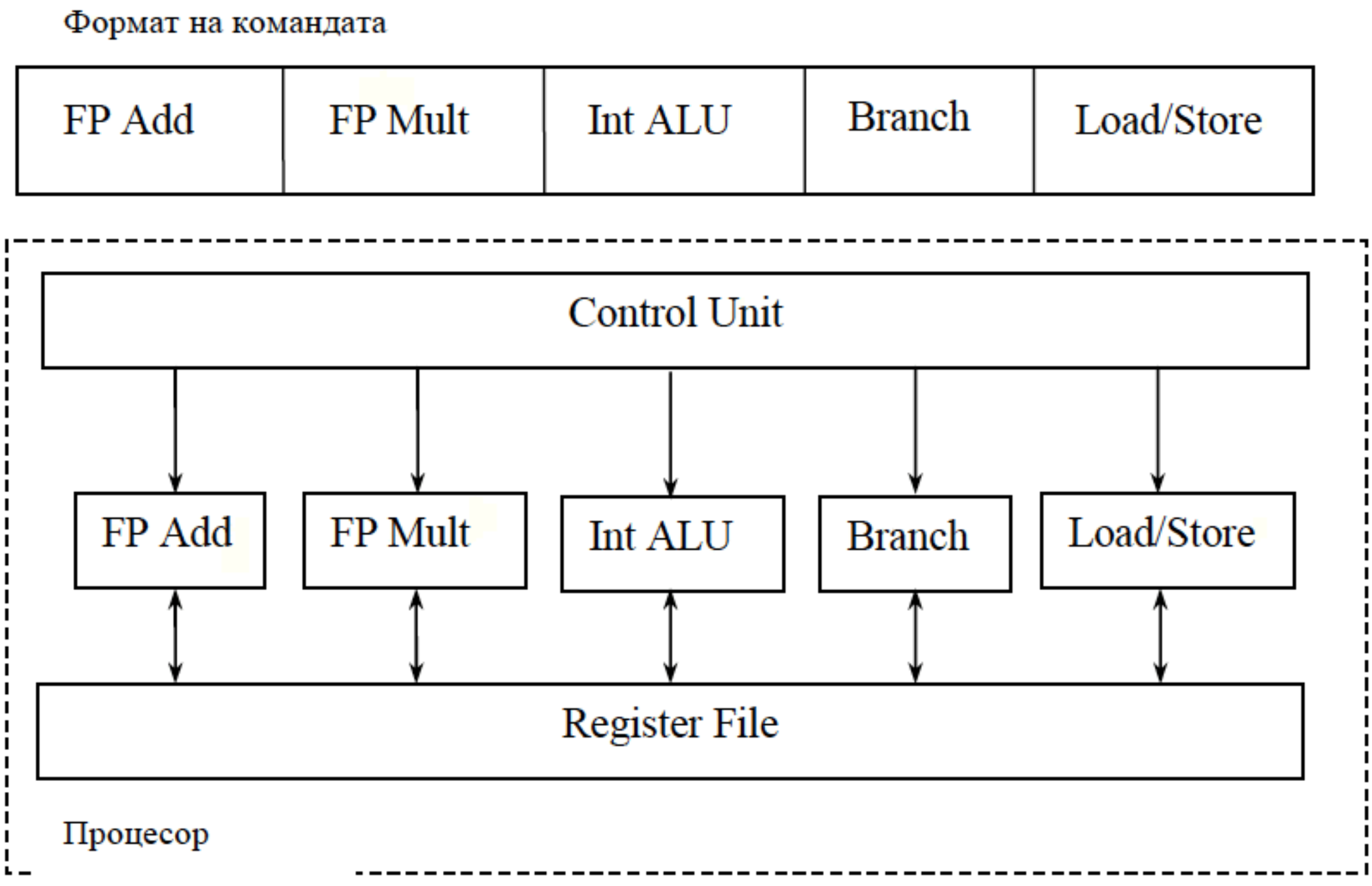
L1D$ *- кэш-память данных 1го уровня*

L1I$ *- кэш-память инструкций 1го уровня*

L2$ *- кэш-память 2го уровня*

CU *- устройство управления*

AAU *- устройство предварительной подкачки данных*



Фиг.5.2 Структура на хипотетичен процесор с **VLIW** архитектура

Процесорът с **VLIW** архитектура притежава следните характеристики:

- Единствен централен контролер издава във всеки такт една дълга команда, т.е. процесорът има един единствен програмен брояч и така се реализира обработка от тип **SIMD**.
- Всяка дълга команда инициира няколко елементарни и независими операции. Това става, като всяко поле директно управлява едно ФУ.
- Всяка операция заема предварително известен брой цикли.
- Всяка операция може да бъде изпълнена чрез конвейер.

Пример. *Даден е следният програмен фрагмент записан на езика С, който трябва да се преобразува в свръхдълги команди за изпълнение.*

```
int i,j;
real a,b,c,q;
{
c=2*i*(2*a+3*b);
q=(a+в+c)-4*(i+j);
}
```

Най-напред компилаторът преобразува изчислението на двата израза в последователност от "стандартни" асемблерни команди. По-долу е даден един примерен вариант, записан на псевдо асемблер (за по-голяма яснота са въведени помощните променливи t1……t8 и се използва нотация характерна за езиците от високо ниво).

```
Load a
Load b
t1=2*a
t2=3*b
t3=t1+t2
Load i
Load j
t4=2*i
c=t3*t4
Store c
t5=i+j
t6=4*t5
t7=a+b
t8=c+t7
q=t8-t6
Store q
```

За да се съставят свръхдългите команди трябва да е ясна структурата на процесора. Ще предполагаме, че той се състои от две устройства за целочислена аритметика (INT1, INT2), две устройства за реална аритметика (FP1, FP2) и две устройства за трансфер на данните между паметта и регистрите (LS1, LS2). Едно примерно опаковане на командите, направено от компилаторът е дадено в Таблица 5-1.

Таблица 5-1.

LS1	LS2	INT1	INT2	FP1	FP2
Load a	Load b				
Load i	Load j			t1	t2
		t4	t5	t3	t7
		t6		c	
Store c				t8	
				q	
Store q					

И така, изпълнимата програма за хипотетичния **VLIW** процесор съдържа 7 свръхдълги команди (думи). Прави впечатление, че не всички ФУ се използват заедно, което говори за по-ниска ефективност. Действително, ако се предположи, че времената за изпълнение на една асемблерна команда и една дълга команда са равни (с първо приближение това е вярно), то лесно може да се определи коефициентът на бързодействие по формула 3.1:

$$S = \frac{T_1}{T_N} = \frac{16}{7} = 2.29\,,$$

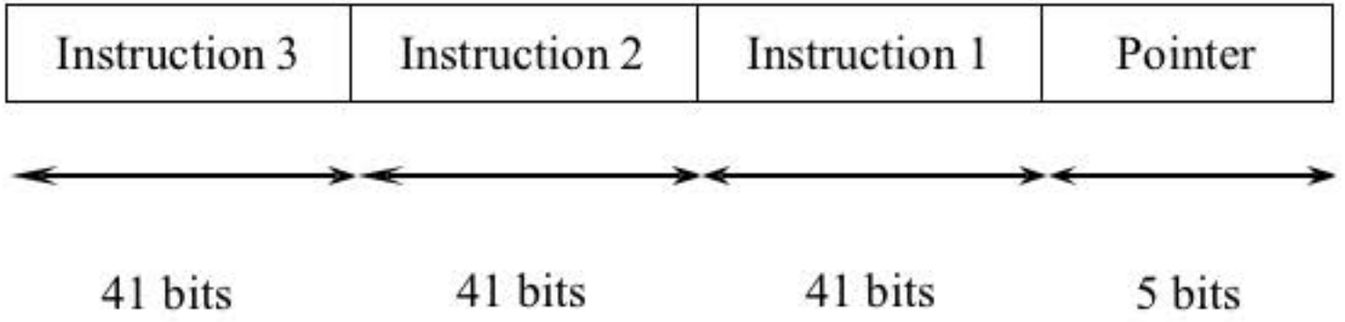
а от тук и ефективността:

$$E = \frac{S}{N} = \frac{2.29}{6} = 0.38\,.$$

Този подход се реализира в различни компютри. В недалечното минало това бяха FPS 120B(L), FPS 164 [1], ИЗОТ 2001С, в компютрите от фамилията TRACE на фирмата Multiflow Computer Inc. [7], ELI 512 разработка на Йелския университет (512 битова дума), QA 1 и QA 2 [2] разработка на университета в Киото (1024 битова дума) и др. Фирмата Transmeta реализира идеите на **VLIW** архитектура в своите процесори ориентирани за използване в преносими компютри [5]. Процесорите от фамилията Crusoe TM5700 и TM5900 имат 128 битова дума и изпълняват до 4 команди от типа x86, а по-новите разработки - Efficion TM8800 имат 256 битова дума и изпълняват 8 команди от типа x86. От началото на 1993г. Intel и Hewlett-Packard разработват архитектура на **VLIW** процесор - Itanium (IA-64), който трябва да поддържа съвместно командите за x86s на Intel и за Precise Architecture на Hewlett Packard [3,4].

Intel официално анонсира Itanium през май 2001, а от юли 2002 и второ поколение процесор - Itanium 2. Понастоящем фамилията Itanium 2 включва McKinly, Madison, Montesito, Hondo (поддържан изключително от HP) и Deerfield. [9]. Архитектурата му е 64 битова и от тук означението IA-64.

Нека накратко да разгледаме архитектурните особености на Itanium-2 от гледна точка на обсъжданата тема. Intel определя архитектурата IA-64 още и като **EPIC** (**E**xplicit **P**arallel **I**nstruction **C**omputing). Както се фижда от фиг.5-3, форматът на командата е с фиксирана дължина от 128 бита и съдържа до три команди от типа x86.



Фиг.5-3. Пакетиране на командите в IA-64 архитектурата

Полето "указател" служи да посочи вида на опакованите команди. Неговите 5 бита предлагат 32 възможности, но се използват за сега само 24 от тях. ФУ са от типа: 6 целочислени устройства, 2 устройства за плаваща аритметика, 2 устройства за четене/запис от/към паметта и 3 за преходите. Конвейерът е 8 степенен. Процесорът може да обработва 2 EPIC команди за такт.

В Таблица 5-2 са обобщени разликите между "класическата" архитектура на Intel -x86 и новата - IA-64.

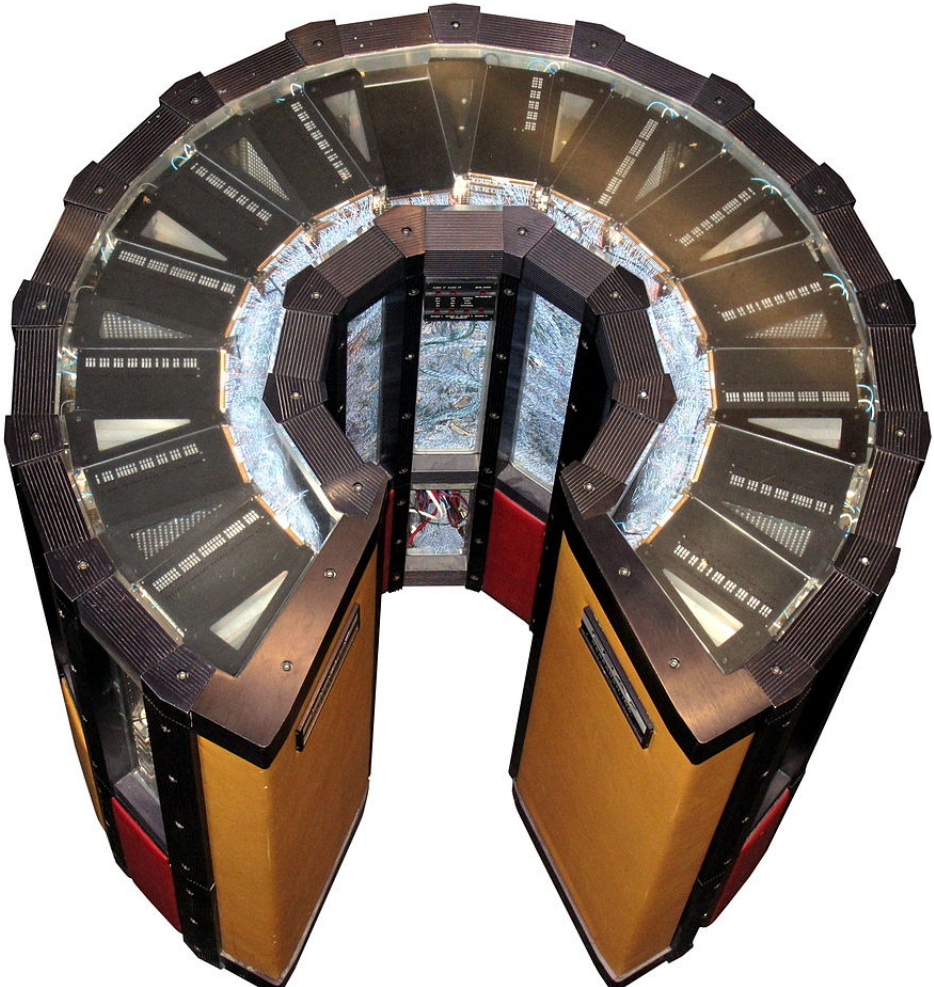
Таблица 5-2

	Архитектура x86	Архитектура IA-64
1.	Използва команди с променлива дължина, обработвани по една за "определено" време.	Използва команди с фиксирана дължина, съдържащи група от по три „прости" (x86) команди.
2.	Преподрежда и оптимизира командния поток по време на изпълнение.	Преподрежда и оптимизира командния поток по време на компилация.
3.	Опитва се да предскаже кой път на прехода ще бъде взет и предварително изпълнява командите, принадлежащи на предсказания път.	Всеки път когато се изпълнява команда за преход, заедно се изпълняват двата пътя на прехода и след това се отхвърля резултата, който не е необходим. Тази техника е позната под името **спекулативно изпълнение**.
4.	Зарежда данни от паметта само когато е необходимо.	Спекулативно зарежда данни преди тяхната необходимост.

Има два подхода за изграждането на матричните процесори.

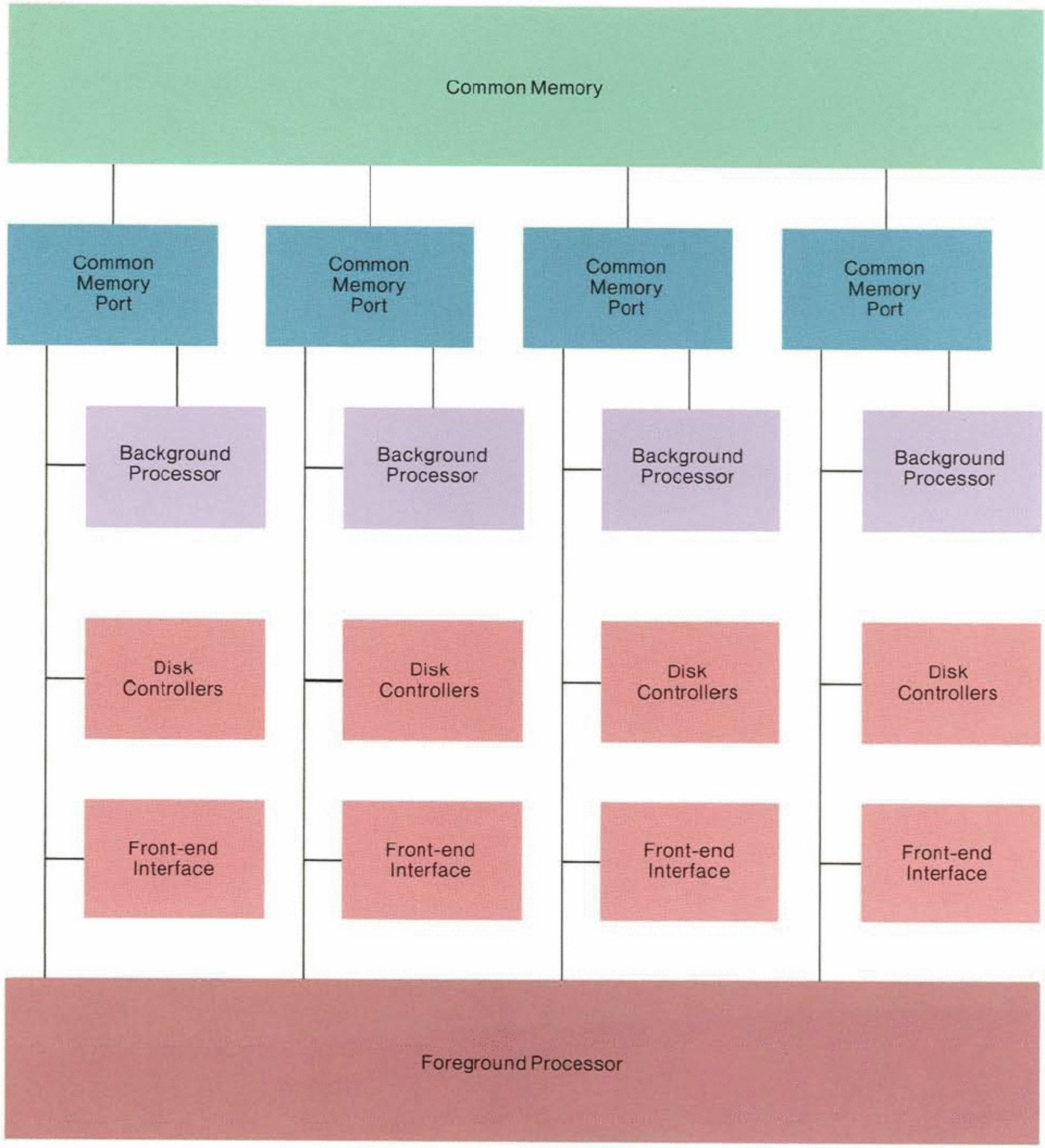
• Първият е чрез съвместяване по време (конвейеризация) на изпълнението на командите, обработващи векторите. Обикновено, процесорите въплъщаващи този подход се наричат векторни процесори, а компютрите, които са базирани на тях - векторни компютри. Примери за такива компютри (процесори) са: CRAY-1, CRAY-2, CRAY X-MP, CRAY CS 6400, Fujitsu VP-200, Hitachi S810, Hitachi S820, FPS 120L(B), FPS 164, IBM 3090 и др. [1-5]. Първите успешни векторните процесори бяха **CDC** Cyber 100 (**C**ontrol **D**ata **C**orporation) и TI ASC (**T**exas **I**nstruments **A**dvanced **S**cientific **C**omputer), които използваха адресация памет-памет за достъп до данните. За първи път в известния CRAY-1 (лансиран през 1974 г.) се въвеждат 8 векторни регистри, всеки съхраняващ 64 думи от по 64 бита всяка. Компютрите на Cray Research бяха изключително успешни и името CRAY стана синоним на суперкомпютър. Различни японски компании (Fujitsu, Hitachi, NEC) също предложиха регистрово ориентирани архитектури решения подобни на тези на Cray Research.

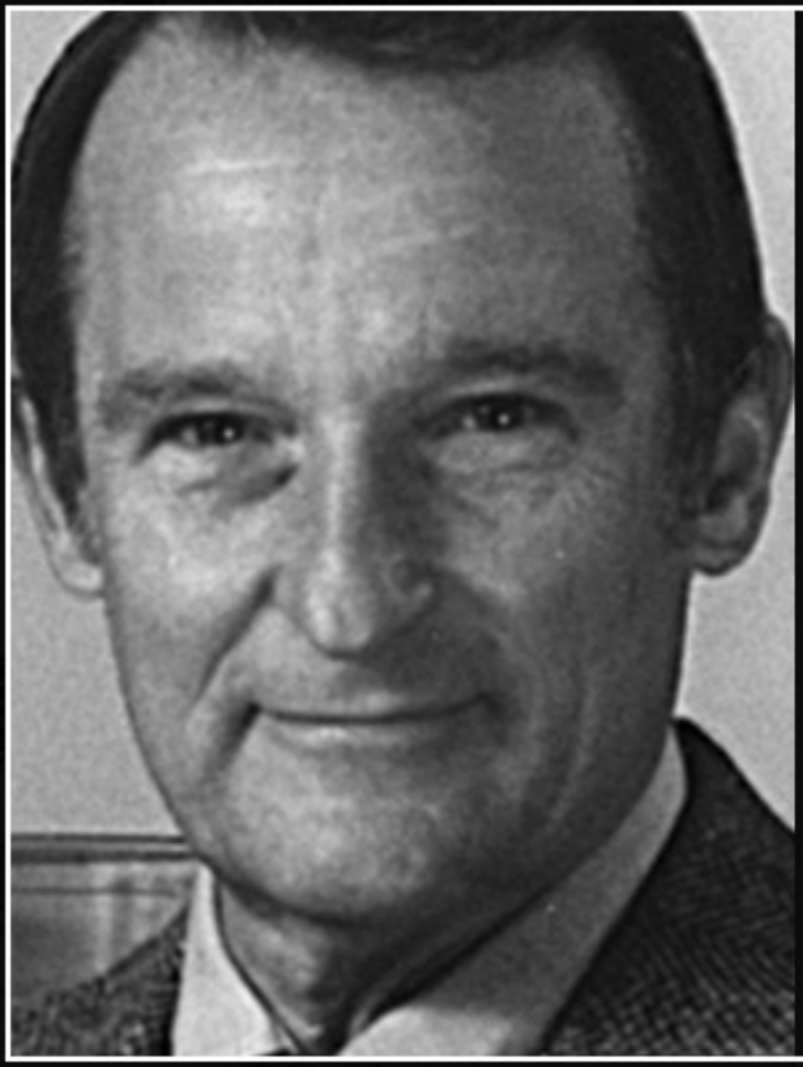
• Вторият подход на изграждането на архитектурата се базира на пространственото повторение на изпълнението на командата. За целта се използват множество процесори, свързани по определен начин, най-често във вид на матрица. Този тип компютри спадат към **SIMD** класа по класификацията на Флин (виж тема 3) и техни представителиса Staran, DAP на ICL, Exemplar 1200/CD 2 или Exemplar1200/XA 8 на Convex, SP2 на IBM, Power Challenge на Silicon Graphics Inc.и др.



Векторен суперкомпютър „Cray–2“:

- От 1985 до 1990 г. най-бързият компютър в света
- Върхова производителност: 1,9 GFLOPS
- Състои се от четири 64-битови векторни процесора
- Елементна база: Емитерно-свързана логика (ECL) / 5 V
- Тактова честота: 244 MHz (тактов период: 4,1 ns)
- Общ обем на паметта: до 2 GB
- Потребление на електрическа енергия: до 195 kW
- Захранван от 2 трифазни 480V / 60→400Hz умформера
- През всяка захранваща шина протича до 2200 A ток
- Охлаждане чрез потапяне в „FC-74“ и маслени помпи
- Габарити: 1143 mm височина, 1346 mm „диаметър“
- 27-те произведени броя продадени за по 12–17000000$


Common Memory
Common Memory Port
Common Memory Port
Common Memory Port
Common Memory Port
Background Processor
Background Processor
Background Processor
Background Processor
Disk Controllers
Disk Controllers
Disk Controllers
Disk Controllers
Front-end Interface
Front-end Interface
Front-end Interface
Front-end Interface
Foreground Processor



If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?

— Seymour Cray —

Основни данни на някои векторни процесори

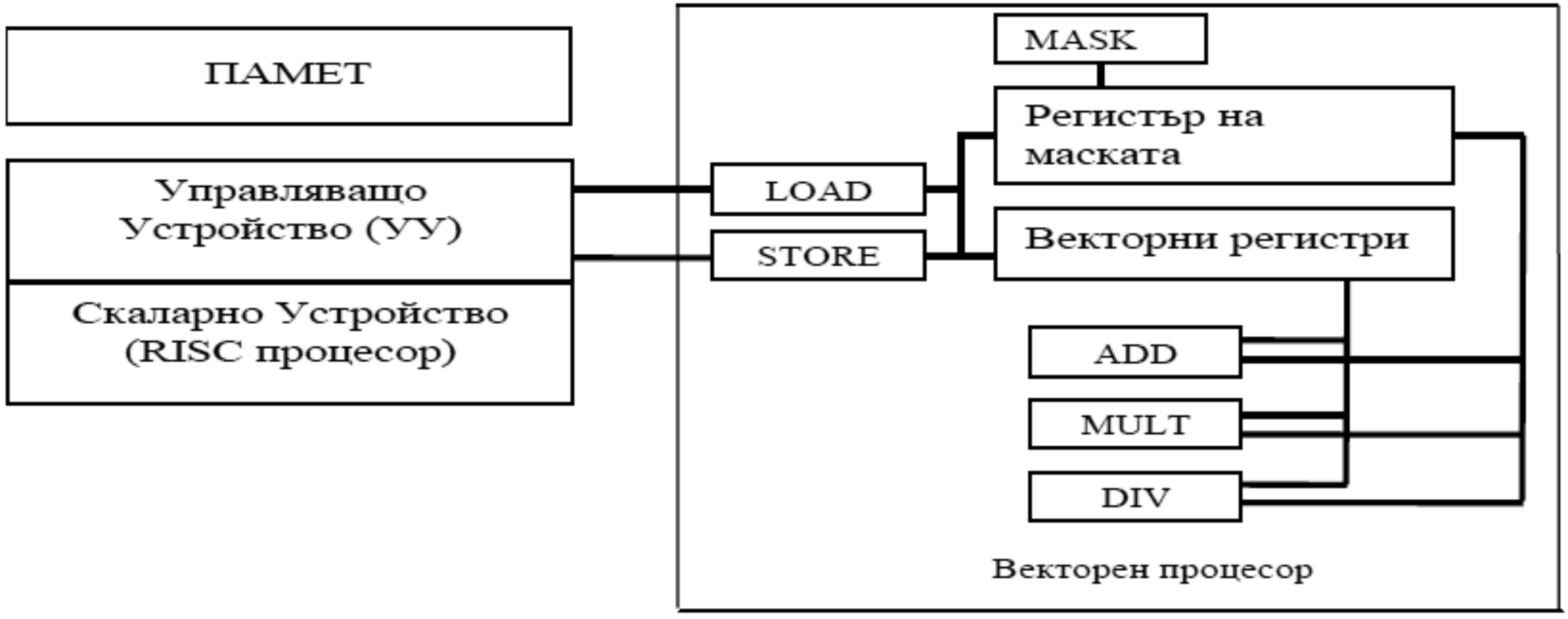
Тип	Пикова производителност	Продължителност на тактовия цикъл	Тактова честота	Максимален обем на паметта	Ширина на лентата на пропускане на паметта
Fujutsu VPP 300/700	2.2 GFLOPS	7 ns	143 MHz	2 GB	18.2 GB
FUjutsu VPP 500	9.6 GFLOPS	3 ns	333MHz	16 GB	76.8 GB
NEC SX4	2.0 GFLOPS	8 ns	125 MHz	2 GB	16.0 GB
NEC SX5	8.0 GFLOPS	4 ns	250 MHz	8 GB	64.0 GB
Alpha EV5	12.0 GFLOPS	1.6 ns	600 MHz	2 GB	1.0 GB

В скаларните процесори много често данните не са част от командата, а се извличат от паметта чрез посочване на адреси. Декодирането на адреса и извличане на данните от паметта отнема известно време. С цел намаляване на това време, повечето модерни процесори използват техниката известна като конвейер за команди – виж тема 5. Векторните процесори използват тази концепция и я развиват. Вместо само конвейер за командите, те също прилагат конвейер върху данните. Например, ако A, B и C са вектори с n елемента, за скаларния процесор сумирането на двата вектора A и B би изглеждало така:

```
for(i=0,i<n-1,i++)
        c[i]=a[i]+b[i];
```

Всяка от тези команди трябва да се декодира и минава през конвейера на скаларния процесор преди да завърши и така не се получава голямо увеличение на скоростта на изпълнение. Но за векторният процесор, тази задача изглежда значително по-различно, а именно:

C=A+B



Фиг.6-1. Обобщена структура на примерен векторен процесор. (ADD, MULT, DIV, MASK, LOAD и STORE са специализирани функционални устройства, работещи на конвейерен принцип)

• Използване на независими модули памет в основната памет за поддържане на конкурентен достъп до независими данни, т.е. да се реализира разслоена памет (виж тема 10).

• Да се използва вътрешна високо-скоростна памет (подобно на кеш-паметта в скаларните процесори).

Тази типова структура на векторния процесор осигурява паралелизъм на три нива:

• На първо ниво е конвейерното изпълнение на аритметичните операции във всяко функционално устройство, така също и между отделните функционални устройства.

• На второ ниво е конвейерното изпълнение на векторната команда.

• На трето ниво, възможно е да се изпълняват паралелно векторна команда и скаларна команда, разбира се ако те са независими една от друга.

Всяка команда, скаларна или векторна, трябва да задава информация за:

- Код на операцията т.е. функция, която се изпълнява.
- Операндите, които се използват.
- Състоянието след изпълнението на командата, което трябва да бъде запомнено (фиксирано).
- Адрес на следващата команда, която се изпълнява.

Както за голяма част от скаларните команди, така и за векторните, последната позиция се определя стандартно.

Общото число на типовете векторни операции не е голямо, но броят на векторните кодове 2-3 пъти превишава броя на съпоставимите с тях скаларни кодове. Причините за това са няколко. Първата е, защото напр. под терминът векторно сумиране се разбира

а) $c_i=a_i+b_i$

б) $c_i=c_i+s$

в) $s=\sum a_i$

г) $s=s+a_i+b_i$.

където a, b, c са вектори, а s е скалар.

С други думи, векторното събиране не е една команда, а група команди, всяка от които извършва операцията сумиране по специфичен начин.

Втората причина за по-големият брой векторни кодове е в наличието на така наречените съставни команди. Това са команди, комбиниращи най-често срещаните последователности от векторни операции, напр.:

а) V=S*V+S*V

б) V=V*V+V*V

в) V=V+S*V и т.н.,

където V са произволни вектори, а S - скалари.

Третата причина за увеличения брой векторни кодове е поради наличието на нови, специфични команди от типа "Сравни два вектора за равенство" или "Провери вектора за нулев елемент" и т.н. Тяхната необходимост се обуславя от причини, разгледани в т.4.3.

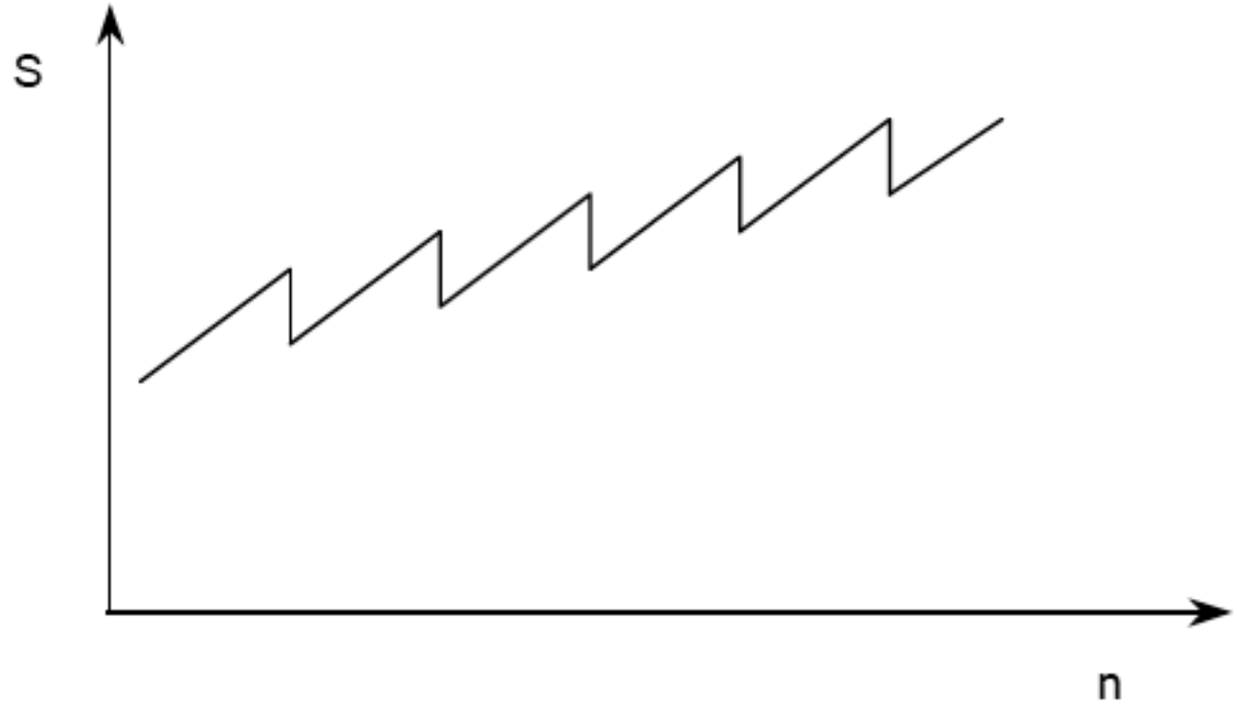
4.2. Адресация на операндите.

Формата на голяма част от векторните команди е триадресен - две полета задават адресите на векторите източници, а третото - на вектора резултат. Адресите, задавани в тези полета са:

- Адреси само на паметта, т.е. адресация от тип памет-памет. Такава адресация напр. се използва в компютрите на CDC Cyber 200/205.

- Адреси само на регистри, т.е. адресация от тип регистър-регистър. Такава адресация напр. се използва в компютрите на Cray Res.

- Смесена адресация, т.е това са адреси както на клетки от паметта, така и адреси на регистри. Такава адресация се използва от IBM във векторния процесор 3090.



Фиг.6-2. Типична зависимост на производителността на векторен процесор от дължината на вектора при адресация от типа регистър-регистър.

Освен информацията свързана с начина на адресиране на операндите, в полетата на операндите трябва да се включи допълнителна информация, която задава:

- Начален адрес на вектора в паметта.
- Размерност на вектора (едномерен, двумерен и т.н.)
- Броят елементи по всяка размерност.
- Типа на данните (цяло число, реално число и т.н.)
- Разположение на данните в паметта.

Има два гранични случая за разполагане на данните в паметта.

- Схемата за разполагане на данните в паметта е повече или по-малко известна предварително.
- Схемата за разполагане се определя по време на обработката.

На тези два случая отговарят и два способа за формиране на адресите: плътна (регулярна) схема за адресация и разредена схема за адресация.

<u>Плътна (регулярната) схема за адресация</u> има от своя страна три разновидности:

- последователна;
- непоследователна, но регулярна;
- подматрична.

Последователна. Ако елемента $\mathbf{V_i}$ на вектора се съхранява в клетка **k**, а елемента $\mathbf{V_{i+1}}$ се съхранява в клетка **k+1** и т.н. се казва, че адресацията е последователна. Тази идея може да се разпространи върху матрици по следните два способа. При съхранение по редове всеки ред на матрицата **A(n,m)** заема **m** последователни клетки, самите **n** реда се съхраняват също последователно. По такъв начин елементът $\mathbf{A_{ij}}$ се намира на разстояние **(i-1)*m+j-1** клетки, от клетката определена за $\mathbf{A_{11}}$. По аналогичен начин е съхранението по стълбове. Първият начин се прилага в езика C, а вторият - в езика Fortran.

Непоследователна, но регулярна. Такъв случай възниква, когато е нужен стълб на матрицата, съхранявана по редове или обратно. На фиг.6-3 е дадено разположение на елементите от редовете на **A** в последователни клетки от паметта.

k	A(i,j)
	.....
	A(i,m)
	A(i+1,1)
	.....
k+m	A(i+1,j)
	A(i+1,j+1)
	.....

Фиг.6-3. Разположение на елементите матрицата **A** в паметта

Подматрична. Третият метод е производен от първите два. Да предположим, че съхраняваме матрицата **A(n,m)** по редове и е необходимо достъп до подматрица с размери **pxq**. За да се получи достъп до тези данни, трябва **p** набора по **q** последователни адреса, при начало на всеки набор с **m** единици по-голям от предидущия. Това съответства на четене на **q** думи и следващите **n-q**, да се пропуснат. Включването на такъв способ за адресация в генератора на адресните вектори изисква във векторната команда да се включат допълнителните параметри **m** и **m-q**.

Разредени схеми на адресация. Плътните (регулярни) схеми на адресация не са подходящи или са неефективни когато:

- Операциите зависят от данните.
- Обработват се разредени вектори.
- Косвено се преглеждат таблици.

Във всички тези случаи се прилагат разредени схеми на адресация. Двете най-разпространени средства са: двоични вектори и векторни индекси. Тяхната основна полза е, че в естествена форма се управлява кои елементи от обработвания вектор участват във векторните операциите. Съществуват три функции за управление: селективно запомняне, свиване и разширение.

Данни	Двоичен вектор	Резултантен вектор
1.1	1	1.1
2.2	0	4.4
3.3	0	5.5
4.4	1	8.8
5.5	1	
6.6	0	
7.7	0	
8.8	1	
9.9	0	

$\longrightarrow$

Свиване

Фиг.6-4. Функцията “свиване” при работа с вектори

Векторни индекси.

Това са вектори, съставени от числа, използувани или непосредствено като адреси на паметта, или като изместване по отношение на някакъв указател на паметта.

Пълната реализация на индексните вектори в общия случай е още по-сложна в сравнение с реализацията на двоичните вектори, защото генератора на векторни адреси трябва да прави два достъпа до паметта - за вектора на индексите и за данните.

4.3. Съхранение състоянието на процесора след изпълнение на командите

Векторните команди трябва да оперират с признаци подобни на скаларните операции, но понеже те работят с многоелементни данни, при формиране на статуса тука се използват кардинално различни способи. Предложени са следните способи (За да се изясни тяхната същност ще се даде пример с фиксирането само на нулев резултат).

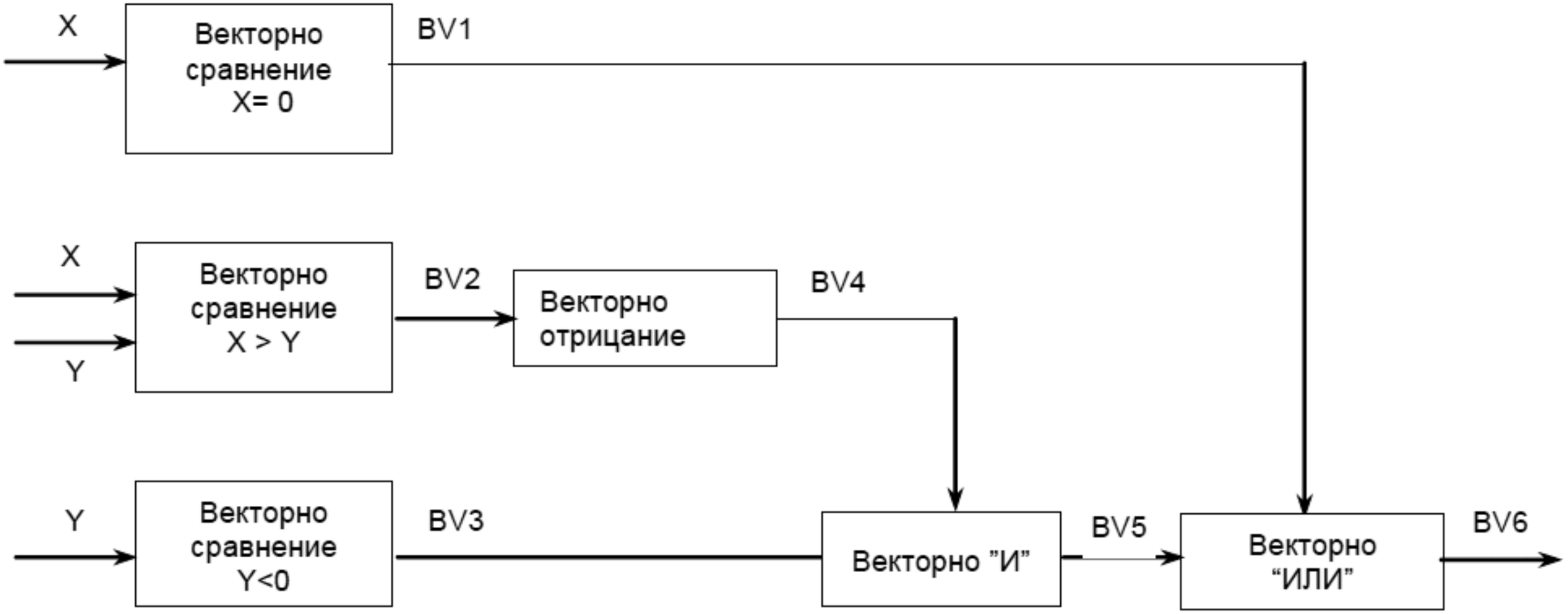
- Да се разширят съответните полета в регистъра на състоянието на векторния процесор. В тези нови полета да се съхранява допълнителна информация, свързана с специфичната обработка на един вектор. Един от възможните начини е в тези полета да се съхраняват признаци за това: един от резултатите е равен на нула; някои от резултатите са равни на нула; всички резултати са равни на нула. Друга възможност е да се регистрира броя нули и мястото, където се е срещнала първата от тях при обработката на вектора и т.н..

- да се формира вектор с кода на признаците, вследствие на което всеки елемент на изхода ще има не само значение, но и собствен код на признаците. За разлика от предходния способ, който изисква регистри за съхраняване на кода на състоянието, тук единствената възможност е формирания вектор на състоянието да се съхранява в паметта.

- Нищо не се запомня. Това е най-разпространеният способ. Вместо това, набора команди се допълва с команди от типа "сравни векторите за равенство", "провери вектора за отрицателно число" и т.н. Всяка от тези команди приема входните векторни операнди и изработва на изхода двоичен вектор.

Пример: *Да се намерят всички значения на елементите на вектора* ***X****, които са равни на 0 или отрицателни, но не превишават съответните значения на елементите на вектора* ***Y****.*

Структурата на програмата е дадена на фиг.7-5.



Фиг.7-5.Структура на векторната програма

За да се тества работоспособността на програмата са зададени конкретни стойности векторите **X** и **Y.** Получените резултати – стойностите на двоичните вектори **BV1**…**BV6** са посочени в таблица 6-1 заедно със стойности на входните векторите **X** и **Y.**

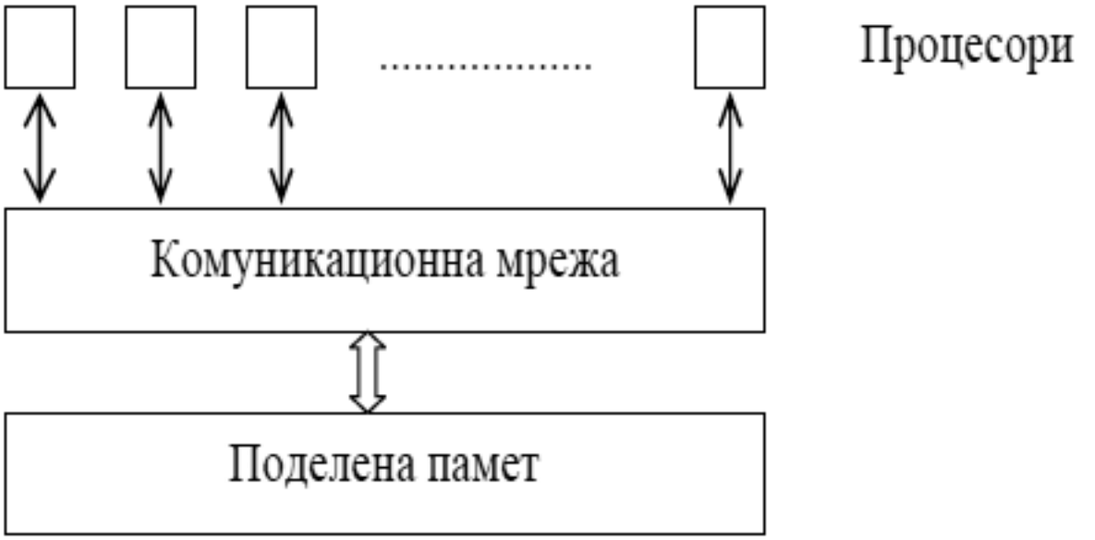
Таблица 6-1

X	Y	BV1	BV2	BV3	BV4	BV5	BV6
0	0	1	0	0	1	0	1
-5	-3	0	0	1	1	1	1
-1	-2	0	1	1	0	0	0
3	1	0	1	0	0	0	0
4	0	0	1	0	0	0	0
0	2	1	0	0	1	0	1
-1	-3	0	1	1	0	0	0

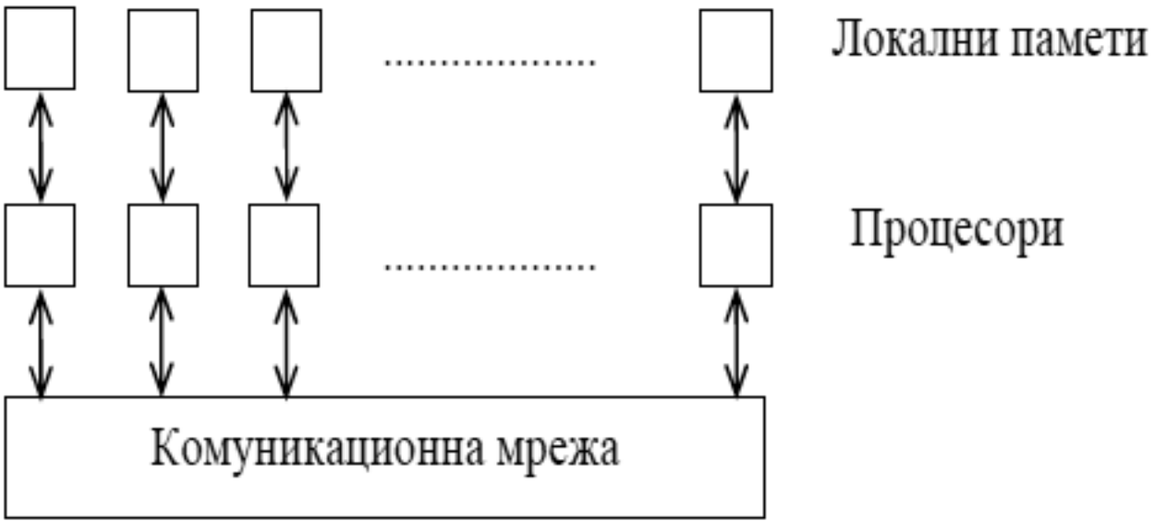
В тема 3, беше посочено, че една от възможните класификации на паралелните компютри се базира на степента на свързаност на процесорите. От тази гледна точка компютрите биват:

- силно свързани;
- слабо свързани.

При силно свързаните системи, няколко процесора, чрез комуникационна мрежа (виж тема 8) директно комуникират с паметта и останалите устройства в системата. При слабо свързаните системи, паралелният компютър се състои от локални компютри, обединени в локална мрежа. Първите системи са познати още като компютри с обща памет – **SMP** (**S**ymmetrical **M**ultiple **P**rocessors), а вторите като компютри с разпределена памет **MMP** (**M**assive **P**arallel **P**rocessors). На фиг. 7-1 са дадени обобщените структури на двете системи.



а) Обобщена структура на SMP компютър



б) Обобщена структура на MPP компютър

Обаче общата памет е причина и за най-сериозния недостатък на **SMP:** когато се добавят нови процесори, трафика по комуникационната мрежа към паметта нараства бързо, докато достигне точка на насищане, т.е. системата е лошо мащабируема. За решаването на проблема се използват:

- кеш-памети включена към всеки процесор;
- подходящо разпределение на данните по модулите памет (виж тема 9).

Използването на кеш-памет поражда друг проблем – необходимо е да се осигури кохерентност за данните, използвани от всичките процесори, но получавани от един процесор. Проблемът е известен като съгласуване на данните в паметта (cache coherence problem) и се дискутира в тема 9.

Обобщено може да се каже, че тясно място в тези компютри се явява недостатъчната пропускателна способност на каналите за връзка на процесорите към паметта, което и ограничава максималния брой процесори.

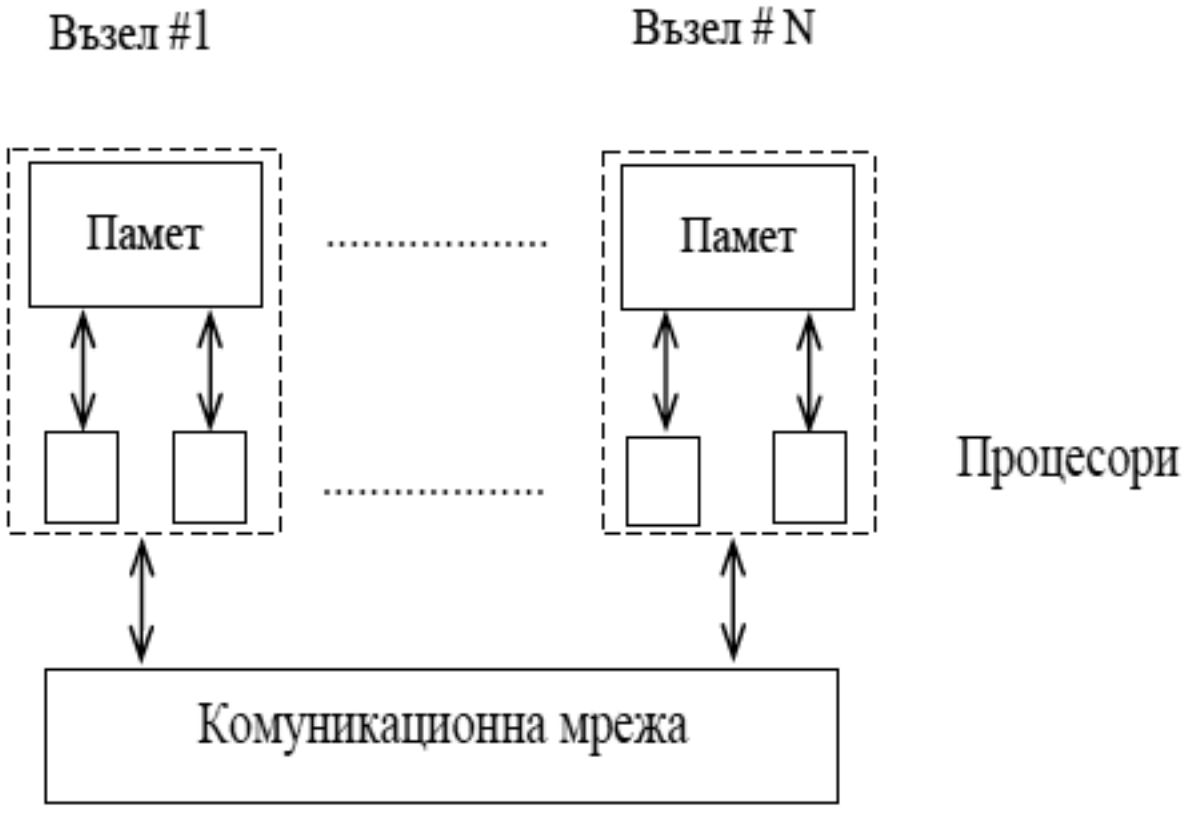
Типични представители на този клас компютри са суперкомпютрите на Cray Research Inc., Silicon Graphics Inc., Hewlett-Packard и др. По същество разгледаните в предходните теми компютърни архитектури спадат към този клас компютри.

необходимо да се осигури ефективно взаимодействие между процесорите, т.е. да се въведе обмен на данните, разположени в различните памети. Това се реализира с помощта на по-бавните входно-изходни операции на процесора. Практически единствения способ за програмиране на този клас системи е чрез използване на обмен на съобщения. Известни са две технологии – **PVM** (**P**arallel **V**irtual **M**achine) и **MPI** (**M**essage-**P**assing **I**nterface), чието приложение не винаги е просто. Благодарение на големия брой съвместно работещи процесори е възможно да се реализират системи с масов паралелизъм и да се осигури мащабируемост на архитектурата. Същевременно компютърните архитектури с разпределена памет не позволяват да се изпълняват съществуващите програми с висока скорост, а преработката на програмите отнема време и струва скъпо. Много често е необходимо да се разработят нови паралелни алгоритми.

Компютрите произвеждани от такива компании като Intel Scientific Computers - **iPSC** (**I**ntel **P**ersonal **S**uper **C**omputer), IBM (напр. Scalable Power Parallel System), NCUBE/ten NCUBE Corporation, компютрите Connection Machine на фирмата Thinking Machines и др. са представители на този клас паралелни компютри.

От изложеното по-горе става ясно, че двата класа имат свои достойнства, които постепенно преминават в техни недостатъци. Може ли да се обединят достойнствата на двата класа компютри? Едно от възможните направления е проектирането на компютри с архитектура **NUMA** (**N**on **U**niform **M**emory **A**ccess).

Един от първите компютри с тази архитектура е бил Cm*, разработен в средата на 70-тях години в университета Карнеги Мелон [13]. След него следват компютри като NUMA_Q на IBM, Alpha Wildfire на Compaq, MIPS64 на SGI и др.[11]. На фиг.7-2 е показана примерна архитектура на такъв компютър. Компютърът се състои от набор от възли, съединени помежду си чрез комуникационна мрежа. Всеки възел обединява няколко процесора (в случая са показани 2 процесора), модул памет, контролер на паметта и периферни устройства (последното не е задължително), съединени помежду си с локална комуникационна мрежа, най-често шина.



Фиг.7-2. Обобщена структура на компютър с NUMA архитектура

От тази особеност и произтича названието на този клас компютри – компютри с нееднакъв достъп до паметта. В този смисъл, класическите **SMP** компютри са с архитектура **UMA** (**U**niform **M**emory **A**ccess), осигуряващи еднакъв достъп за произволен процесор.

Както и за **SMP** компютрите, и за този клас компютри е характерно несъгласуваност на данните на ниво кеш памет. За решаването на този проблем е разработена специална модификация на **NUMA** архитектурата – **ccNUMA** (**c**ache **c**oherent **NUMA**). За целта са разработени множество протоколи, съгласуващи съдържанието на всички кеш памети и програмиста не се грижи за този проблем.

Броят на процесорите в сървърите с **SMP** архитектура е от порядъка на 8-16, докато **NUMA** архитектурата обединява 256 и повече процесори.

В групата компютри с разпределена памет, освен традиционните компютри, посочени по-горе, се включват и кластерните системи. В компютърната литература значението "кластер" се употребява в различен аспект. В частност, "кластерната" технология се използва за повишаване на скоростта и надеждността на сървърите за база данни и Web-сървърите. Тука ние ще обсъждаме само кластерите, ориентирани за решаване на задача от изчислителен характер. В този смисъл кластер означава съвкупност от компютри, обединени в някаква мрежа за решаване на една задача. В качеството на изчислителни възли обикновено се използват достъпни на пазара еднопроцесорни компютри, дву- или четири- процесорни **SMP** сървъри. Всеки възел работи под управлението на свое копие на операционната система. Състава и мощта на всеки възел може да се мени, което дава възможност за създаване на нееднородни системи. Към тази група паралелни компютри спадат проекта Beowulf създаден в NASA, проекта KLAT и KLAT2 [10], а също така и компютрите на SGI от серията Origin и др.

2. Абстрактен модел за изчисление в MPP компютрите

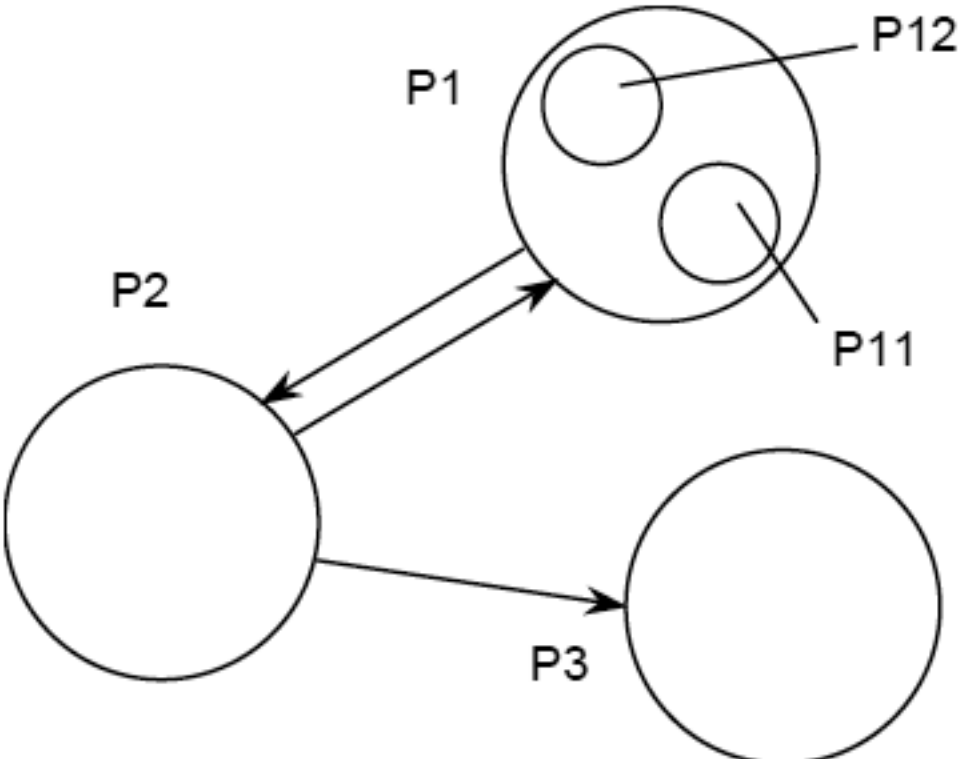
Хоар предлага следния абстрактен модел на изчисления [8]: паралелният изчислителен процес представлява изпълнение на независими конкуриращи се процеси, обменящи данни помежду си чрез един свързващ път, наречен канал. Моделът е известен под името "комуникация на последователни процеси" (**C**ommunicating **S**equential **P**rocesses - **CSP**).

Основните елементи в **CSP** модела са процес и канал.

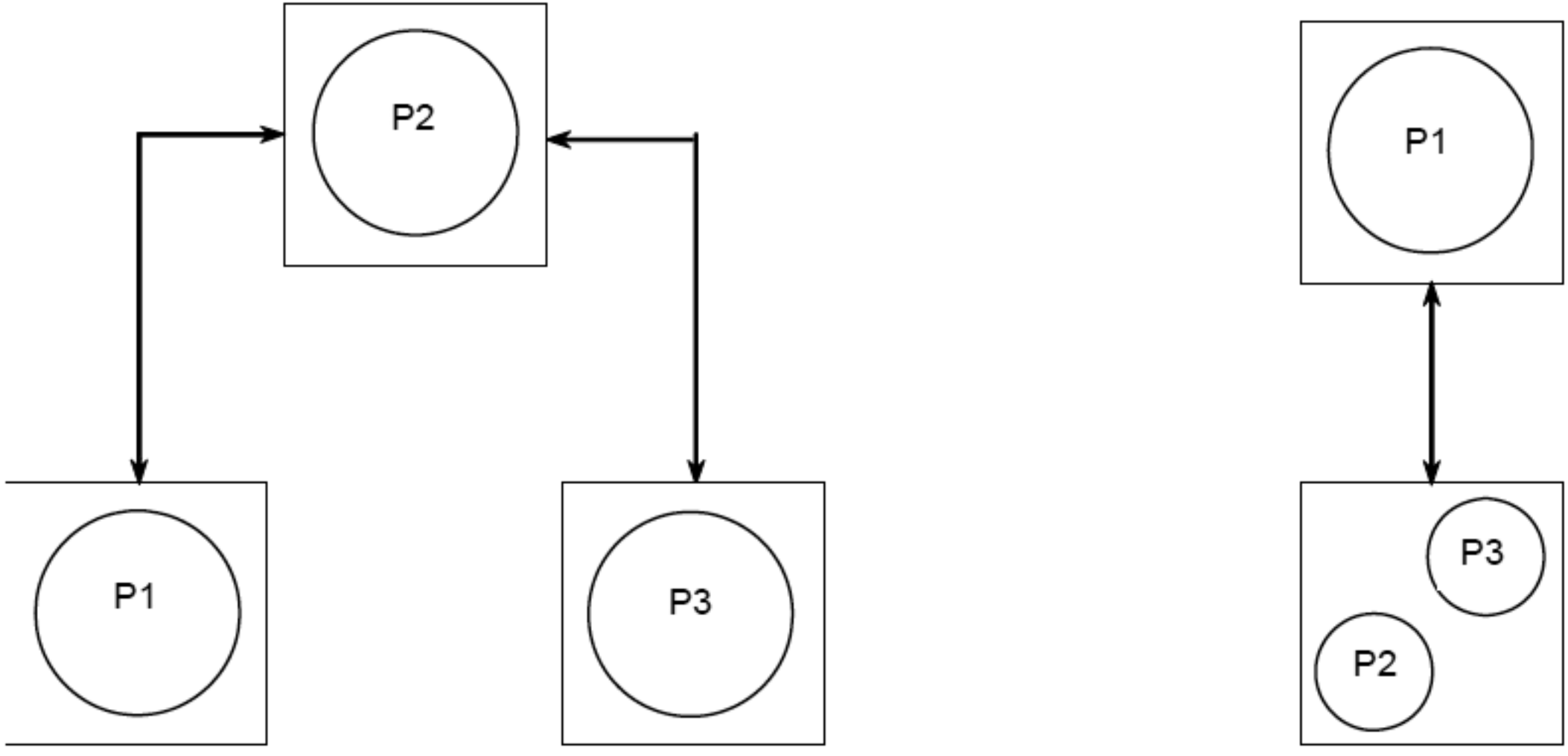
Процес. Процесът описва поведението на дискретен, отделен компонент от приложението. Той може да се състои от други процеси. Процесите могат да бъдат разположени върху произволна мрежа от процесори.

Канал. Каналът осигурява връзката между два процеса. Каналът е от тип "точка - точка" и е еднопосочна връзка, така че той свързва само два процеса. Каналът в **CSP** модела има две функции:

- Осигурява път между независими процеси и така се обменят данни между тях.
- Осигурява синхронизация на комуникацията между двата процеса. Това се осъществява по следния начин. Процесът-приемник трябва да потвърди получаването на данните. В същото време изпълнението на процеса-източник чака получаването на потвърждението и не може да продължи работа.



Фиг.7-3. Прост **CSP** модел.



Фиг.7-4. Разположение на един и същ **CSP** модел върху различни мрежи от процесори.
а) върху три процесора; б) върху два процесора.

съобщения: **P**arallel **V**irtual **M**achine (**PVM**) или **M**assage **P**assing **I**nterface (**MPI**).

Главният въпрос при проектирането на компютри с разпределена памет е ефективното взаимодействие между процесорите, защото за достъп до данните, разположени в не локалните памети е необходимо използване на бавните операции за вход-изход на процесора. Подобни обръщения стават с помощта на съобщения, които се обменят между локалните памети на два процесора. Ето защо понякога, този клас компютри се наричат компютри с обмен на съобщения (message passing). Механизмът на управление на тези съобщения оказва голямо влияние във всички аспекти при разработването на компютъра и програмното осигуряване. Така за осигуряването на взаимодействие между процесорите (което се явява част от операционната система, намираща се във всеки възел) е необходимо наличието на високопроизводителна комуникационна мрежа и подходящо програмно осигуряване.

Нека да разгледаме
решаването на една тривиална задача - умножение на две матрици върху четирипроцесорна система. Тази задача се решава чрез два различни алгоритъма; първият е така наречения вълнов алгоритъм, при който съобщението е с големина един елемент (4 байта) и не зависи от размера на обработваните матрици, а при вторият изчисленията са организирани по-такъв начин, че процесорите работят самостоятелно върху големи сегменти от данни и обменят съобщения с размерност kn^2 елемента, където k зависи от размера на обработваните матрици. Първият алгоритъм реализира така наречения "фин" паралелизъм, а вторият - "груб" паралелизъм. Времената, дадени в секунди, за три различни метода на умножение на матриците, са поместени в таблица 7-1.

Таблица 7-1

		паралелни изчисления	
размерност n	последователни изчисления	фин паралелизъм	груб паралелизъм
10	0.005	0.007	0.003
20	0.038	0.057	0.012
30	0.129	0.200	0.043
40	0.304	0.432	0.081
50	0.593	0.843	0.165

Забележка: Резултатите са от изследвания на Цветан Таслаков, проведени с процесор (транспютър) T805/30 на INMOS, а средата за програмиране е TOOLSET ANSI C.

Програмирането става обикновено чрез написването на отделни програми за всеки процесор. Програмите се свързват чрез операциите от ниско ниво, които предоставя операционната система и са достъпни за програмиста във вид на разширение на последователните езици от типа на FORTRAN, Pascal или C. Като правило, програмите във възлите са копия на една и съща програма. Отделните копия ще се изпълняват правилно, независимо от тяхното разположение в мрежата. Такъв стил на програмиране е известен като **SCMD** (**S**ingle **C**ode **M**ultiple **D**ata) или **SPMD** (**S**ingle **P**rogram **M**ultiple **D**ata).

Още един проблем, който е свързан както с приложната задача (и по-точно с разработването на подходящ алгоритъм), така и с работата на операционната система, е осигуряването на балансирано натоварване. Под балансирано натоварване трябва да се разбира равномерното разпределение на задачата между процесорите, така че да се осигури работа на всички процесори, или на по-голямата част от тях, през цялото времетраене на решението на задачата. Неизпълнението на това изискване или лошото му изпълнение води до по-голям брой комуникации и от тук до снижаване на производителността.

Прехвърлянето на данни, команди и управляващи сигнали през КМ изисква време, което формира така наречените комуникационните загуби. Времето за комуникация, като правило, се прибавя към изчислителното време и така се очертава тенденция за увеличаване на общото време за решаване на дадена задача. Съществуват два основни подхода за съкращаване на комуникационните загуби:

- За сметка на оптимално разпределение на решаваната задача (код и данни) по процесорите и/или по модулите памет.
- За сметка на повишаване на скоростта на работа на самата КМ.

Първият подход е обект на изследване при конструирането на паралелните алгоритми и поради това по-нататък няма да се дискутира, независимо от тясната връзка между паралелната архитектура и паралелния алгоритъм. Вторият подход, от своя страна също се реализира по два начина:

- Чрез апаратни средства с използването на по-бързодействаща логика.
- Подходяща структурна организация на КМ.

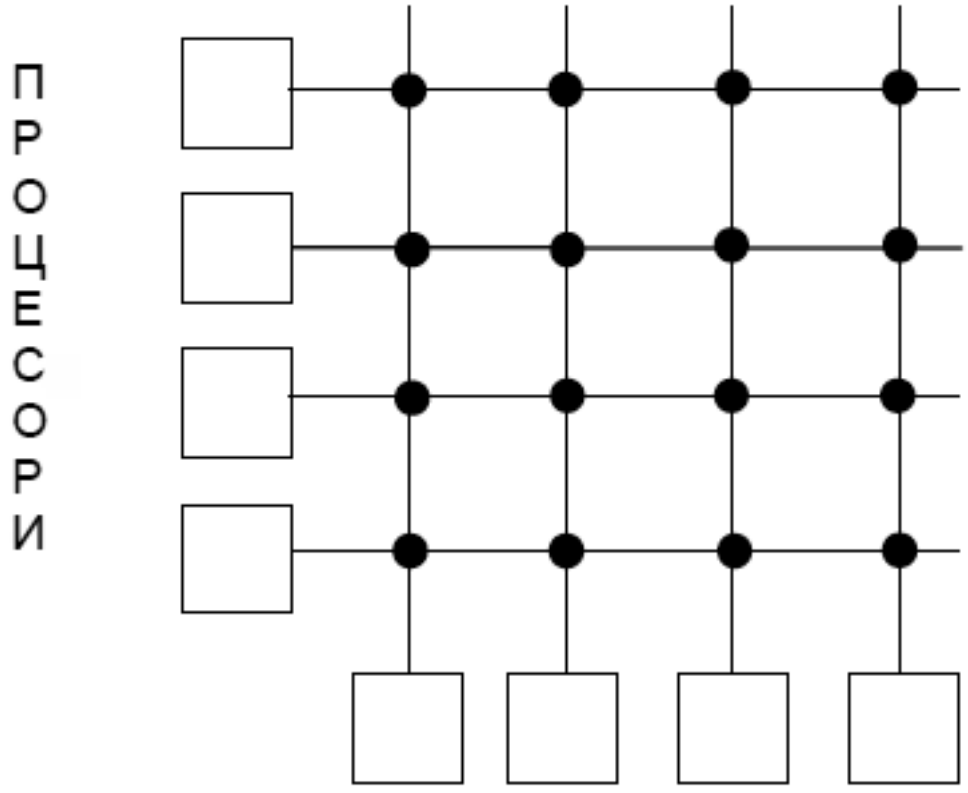
В общият случай КМ представлява MxN многополюсник, входните M полюси на който са присъединени към предавателите, а изходните N полюси – към приемниците. КМ се изграждат от комутиращи елементи (КЕ) и линии за връзка (ЛВ). С помощта на КЕ се осъществява свързването на ЛВ и така се осигурява път за предаване на информацията. По-нататък ще се разглежда само случая когато M=N.

КМ могат да осигуряват комутация:

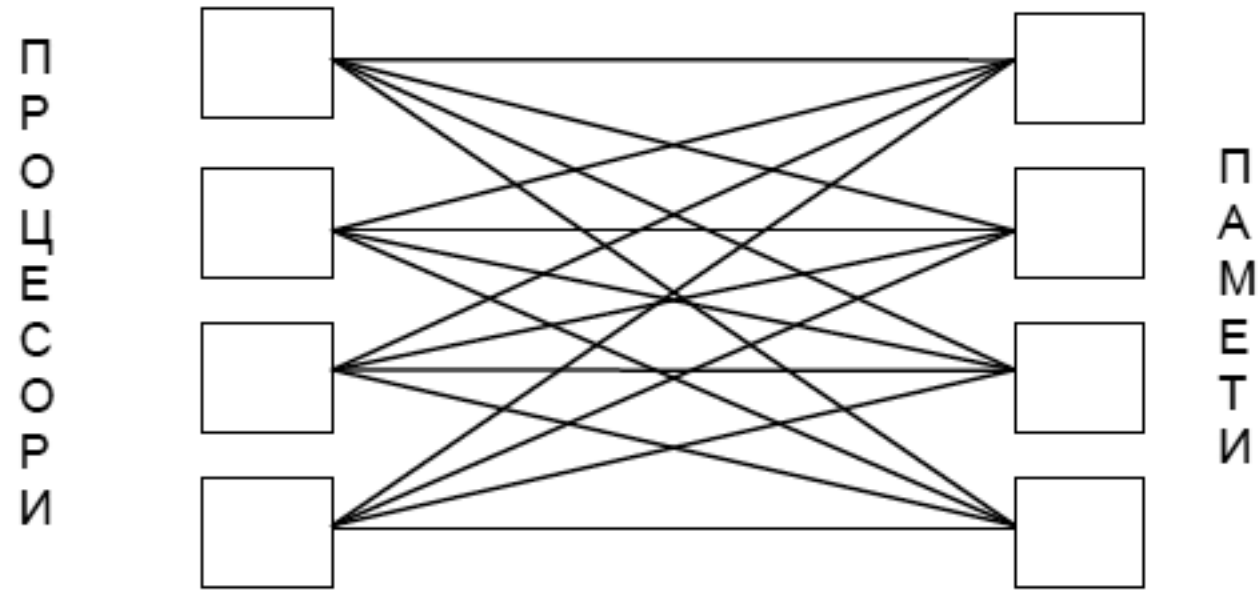
- по пространство;
- по време.

Характерен признак на КМ с комутация по пространство е наличието на индивидуални ЛВ между източниците и приемниците на информацията. При едновременна работа на няколко източника, за всеки от тях се формира принадлежащи само на него ЛВ, свързващи го с приемника и така се установява индивидуално съединение. Типичен представител на този клас КМ е матричния превключвател (cross-bar мрежата) и свързването всеки с всеки (пълно свързаната мрежа - full connecting network) - фиг.8-1. За този клас КМ е характерно висока сложност - $O(N^2)$ и малко време за трансфер - O(1). За големи стойности на N този тип КМ става прекалено скъпа.

При КМ с комутацията по време, няколко двойки предавател-приемник са свързани с обща линия, като всяка двойка я заема по различно време. Типичен представител на този клас КМ е шината. За този клас КМ е характерно ниска сложност - O(1) и голямо време за трансфер - O(N).



а) Матричен превключвател



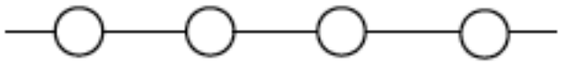
б) Пълно свързана мрежа

Фиг.8-1. Типични представители на КМ по пространство:

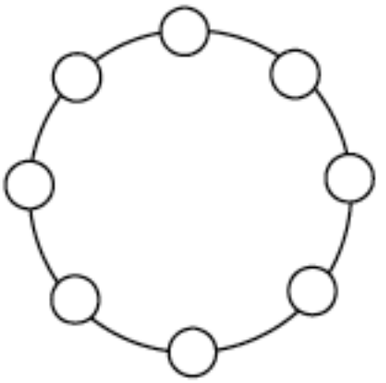
2. Характеристики на КМ

КМ могат да бъдат характеризирани в зависимост решаването на четирите основни въпроса:

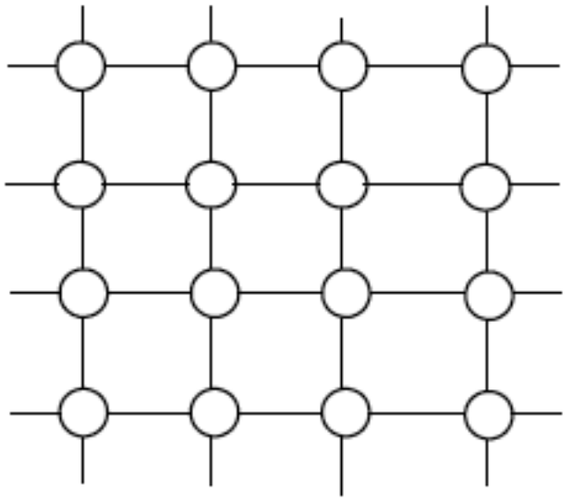
- режим на работа;
- стратегия на управление;
- метод на превключване;
- топология на мрежата.



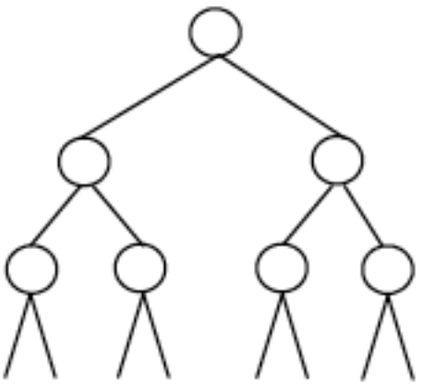
а) линия (конвейер)



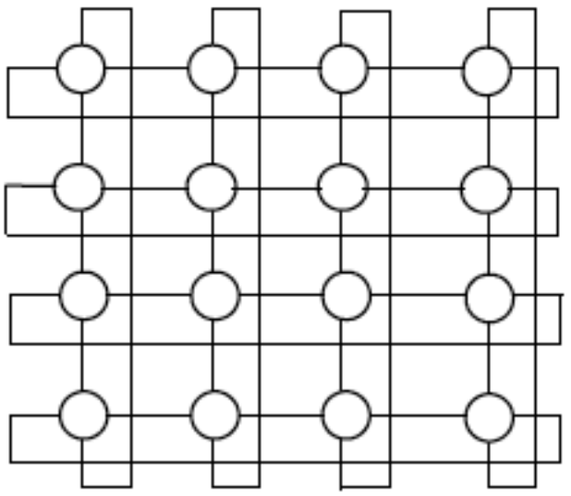
б) кръг



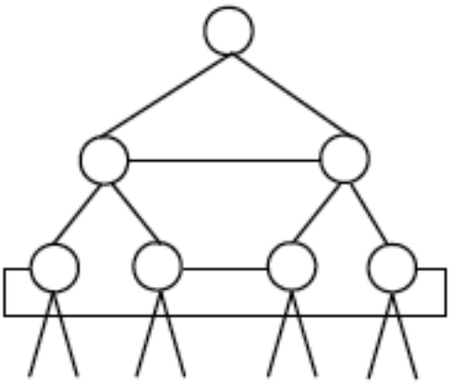
в) двумерна решетка



г) двоично дърво



д) тор



е) хипердърво от тип I

За сравнителна оценка на различните топологии СКМ се използват следните параметри:

Брой на върховете в мрежата - N. Това на практика са процесорите в паралелния компютър.

Диаметър на мрежата - D. Дефинира се като максималното разстояние между произволна двойка върхове в пътя свързващ двата върха. В случая под разстояние се разбира минималния брой на дъгите, свързващ двата върха. Например, за СКМ тип "кръг", D=4, а за СКМ тип "решетка", D=6 и т.н.

Средно разстояние - P. Определя се като средно разстояние на пътищата от всеки връх до всички останали върхове, при предположение, че с еднаква вероятност всеки връх може да бъде предавател, а всички останали - приемници.

Натовареност на линиите за връзка -T. Определя се като брой съобщения за единица време върху една линия за връзка:

$$T = \frac{NxP}{\text{общ брой на линиите за връзка в мрежата}} .$$

Относителна структурна сложност - U. Това е броят на линиите за връзка отнесени към един връх. Когато U=const и не зависи от N, нарастването (намаляването) на СКМ се постига чрез просто добавяне/отнемане на еднотипни КЕ/процесори. От гледна точка на технологията това е изключително примамливо, защото дава възможност за намаляване на общата цена на КМ, чрез използуване на унифицирани компоненти (процесори).

Отказоустойчивост на КМ. В много случаи е важно КМ да съхранява работоспособността си в условията на отказ на отделните компоненти. За тази цел е необходимо да съществуват алтернативни пътища между произволна двойка предавател-приемник, за да бъде заобиколен отказалия компонент. Това напр. е възможно в топология "решетка", но не и в топология "кръг" или "двоично дърво".

За някои от КМ съществуват аналитични изрази за определяне на основните параметри, най-вече на D – виж таблица 8-1, но за по-голямата част от тях определянето им се реализира чрез използването на известен алгоритъм за намиране на пътищата в граф, напр. на Форд или Дейкстра и др.

Таблица 8-1.

Тип КМ	D
Линия (едномерна решетка)	N-1
Кръг	$\left\lceil \frac{N}{2} \right\rceil$
Двумерна (квадратна)решетка	$2(\sqrt{N}-1)$
ILLIAC IV (кръг с хорди)	$\sqrt{N}-1$
Тор	$\sqrt{N}-1$
Двоично дърво	$2(\log_2(N+1)-1)$
Двоичен куб	$\log_2 N$

Ако се придържаме към оптималната маршрутизация на пакетите, то времето за обмен в мрежата е O(D). Във връзка с това, от изключителен интерес е задачата за построяване на плътни графове, съдържащи (при зададени D и U) максимален брой върхове – N_{max}.

Динамичните комуникационни мрежи (ДКМ) осигуряват директна връзка между произволна двойка предавател-приемник в паралелния компютър. Това става чрез промяна на комуникационните връзки посредством реконфигурация на активните КЕ. ДКМ и са еднакво подходящи както за схемна комутация, така и за пакетна комутация. ДКМ могат формално да се обозначат като $F:\{N\}\rightarrow\{N\}$, което се тълкува като размяна F на множеството размествания {N} от N елемента.

Класически пример за ДКМ се явява матричния превключвател. Както беше посочено в т.1, неговият основен недостатък е квадратичното нарастване на броя КЕ в зависимост от броя бходове (изходи) на КМ; с други думи става сложно за реализация, дори невъзможно в някои случаи, на мрежа свързваща няколко стотин или хиляди процесора. Едно от възможните решения на този проблем е да се използва многостъпална мрежа.

Многостъпалните мрежи осигуряват пълна система на връзките в рамките на паралелната структура. Те от своя страна се разделят на:

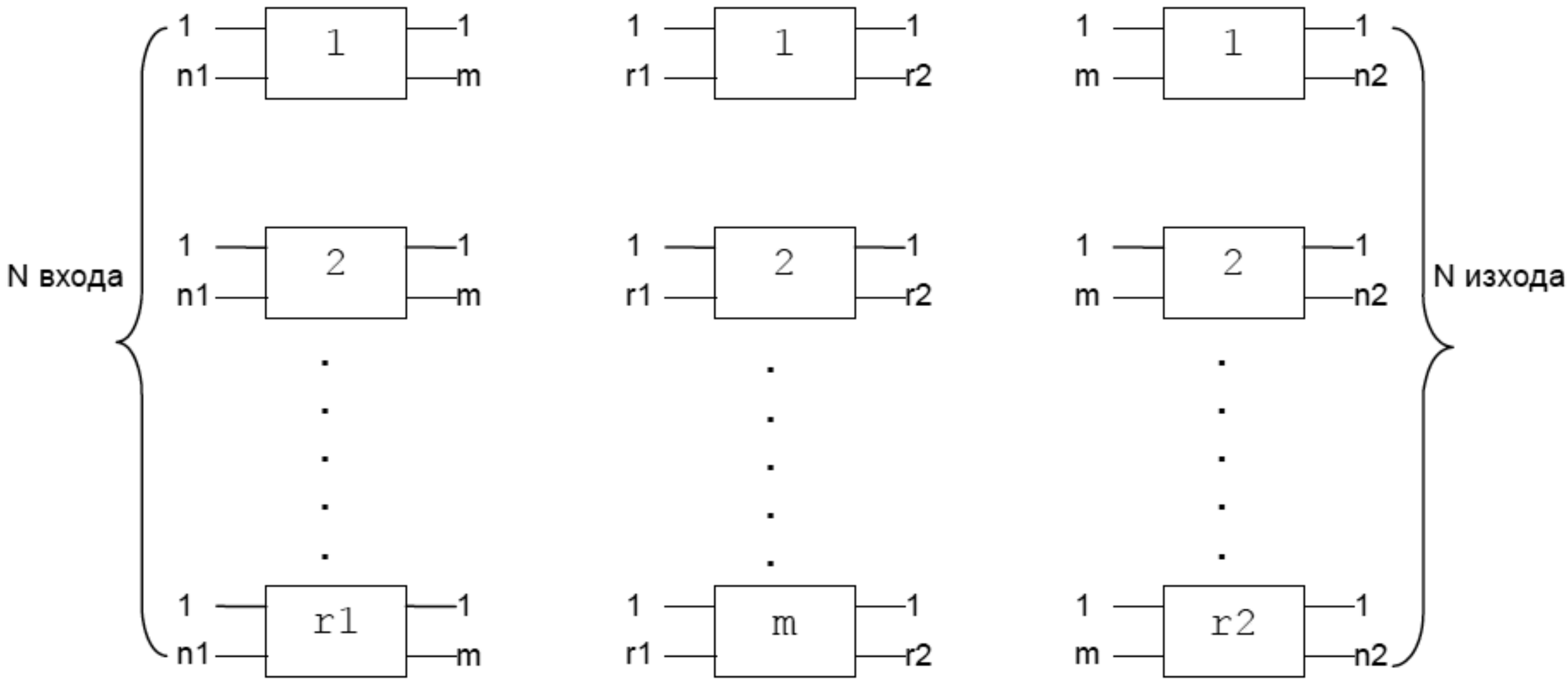
- едностранни
- двустранни.

Едностранните мрежи използват двупосочни ЛВ, а двустранните имат входна страна и изходна страна. Последните от своя страна се разделят на блокиращи, неблокиращи с реконфигурация и напълно неблокиращи.

В блокиращите ДКМ е възможно с присъединяването на дадена една двойка предавател-приемник да предизвика конфликт в използуването на мрежата за друга двойка предавател-приемник. Най-известни представители на този тип ДКМ са манипулатор, делта, омега, флип и др. Напълно неблокиращите ДКМ, както подсказва името им, не страдат от този недостатък. Примери за такива мрежи са мрежата на Клос и пълносвързаната мрежа. В неблокиращите мрежи с реконфигурация също е възможна реализация на съединенията между произволна двойка предавател-приемник, но за това е необходимо да се измени маршрута на връзките за съединените вече двойки терминали. Примери за такива мрежи са мрежата на Бенес, битоналната мрежа и др.

Построяването на многостъпални КМ е свързано с необходимостта от използуване на по-прости (в структурно отношение) КЕ. Това води до увеличен брой стъпала в мрежата, k на брой стъпала в общият случай. Като следствие на това се получава:

- Увеличено време за предаване на съобщенията.
- Усложнен алгоритъм за управление.



Фиг.8-3. Обобщен модел на k-стъпална КМ (k=3).

Първото (входно) стъпало се състои от r_1 КЕ (комутатора) с размерност n_1xm (при това $r_1xn_1=N$), междинното стъпало се състои от m КЕ с размерност r_1xr_2 и последното трето стъпало (изходното) се състои от r_2 КЕ с размерност mxr_2 (при това $r_2xn_2=N$).

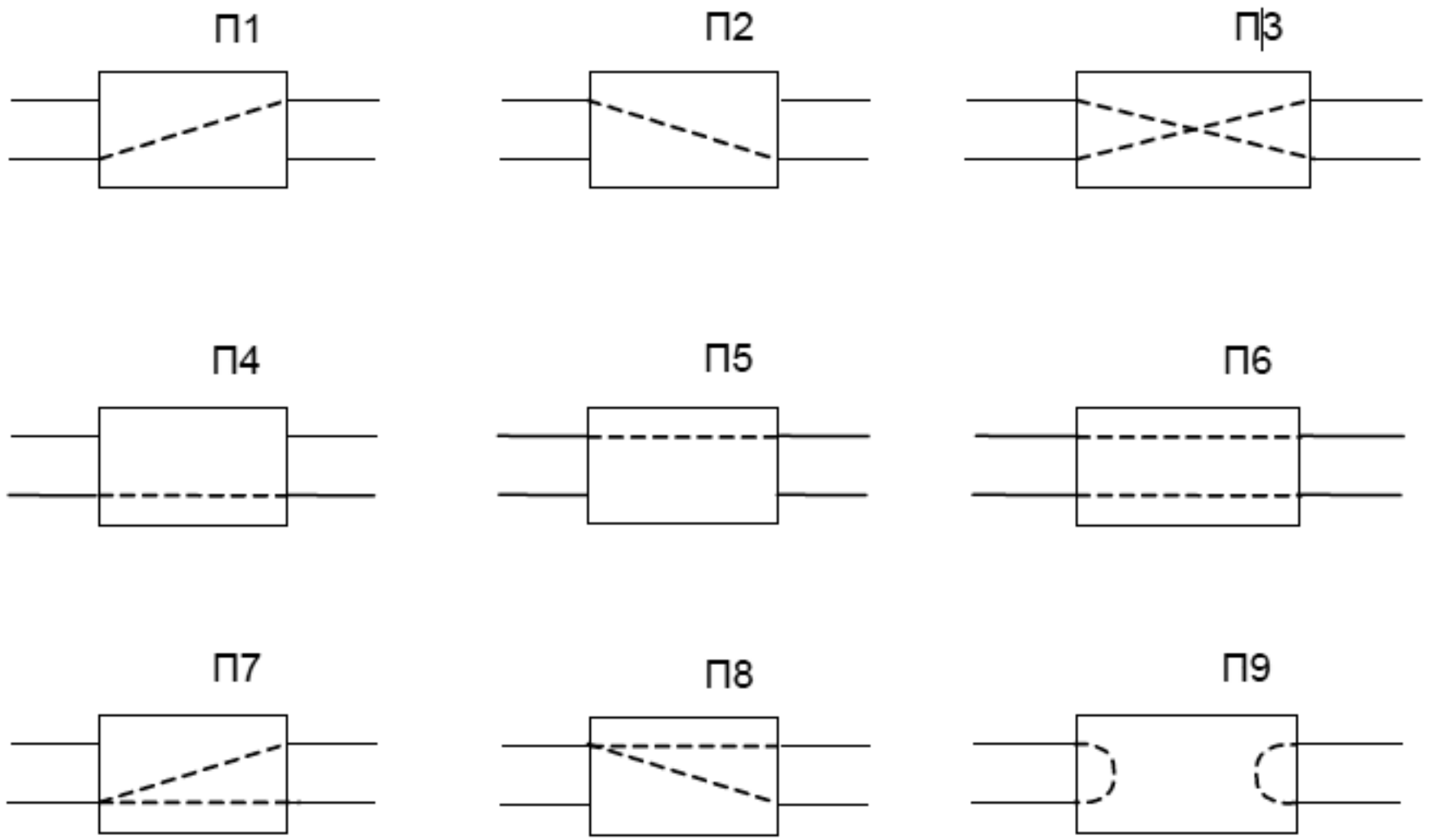
За оценяване на различните топологии ДКМ се използват най-често следните параметри:

Време за предаване. Времето за предаване, с едно първо приближение, може да се счита, че е O(k), т.е. колкото по-малко стъпала има в мрежата толкова по-малки ще бъдат комукационните загуби. Същевременно по-малкият брой стъпала означава, че КЕ в структурно отношение ще бъде по-сложен (за достигане на един и същ брой входове/изходи на мрежата) и по този начин той ще осигурява по-голяма задръжка при преминаване на сигналите през него.

Сегментиране. Това е разделяне на ДКМ на независими подмрежи с различни размери. Всяка подмрежа трябва да има всички възможности за връзки на цялата мрежа от същия тип и размер.

Ширина на лентата на пропускане. Този параметър се дефинира като очакван брой заявки приети за единица време. Тъй като системната шина не може да осигури широка лента на пропускане за голям брой процесори, а матричния комутатор е скъп, то е интересно и важно да се знае как варира лентата на пропускане при различните ДКМ за различните случаи. Най-често тези оценки се получават с помощта на подходящи имитационни модели.

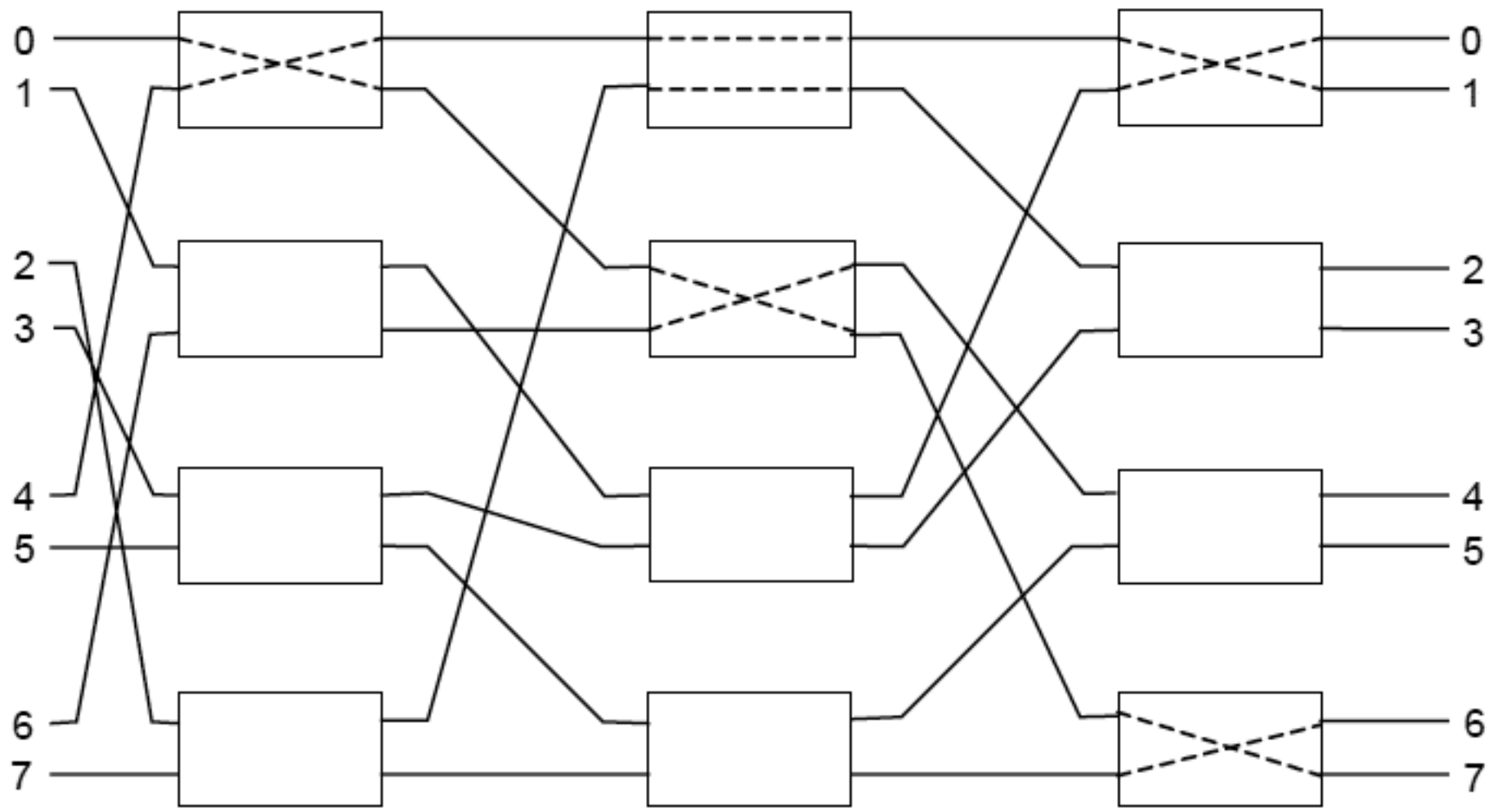
Най-широко разпространение са получили КЕ елементи с два входа и два изхода. Възможните им състояния са показани на фиг.8-4.



Фиг.8-4. Възможни състояния на КЕ от тип 2x2.

Не всички възможни превключвания се използват във всяка ДКМ. Например, в едностранните ДКМ е необходимо така нареченото "трето състояние" – $П_9$. Ако в мрежата е допустимо да се използва режим разпръскване (размножаване) се използват $П_7$ и $П_8$.

Състоянието на КЕ се задава от секцията за управление на динамичната реконфигурация в рамките на локалния или глобалния контролер в зависимост от приложената стратегия на управление – децентрализирана или централизирана. Последният се управлява от предварително изчислен тяг, най-често определен по формулата $T=X \oplus Y$, където X и Y са съответно адреси на предавателя и приемника. На фиг.12-6 е дадена delta мрежата за N=8.



Съвременната тенденция е да се изграждат интелигентни КЕ, които участват активно в координиращите дейности в паралелния компютър, като:

- реализират локалното разпределение на товара;
- регулират трафика в мрежата с цел детектиране и елиминиране на насищането и мъртвото блокиране;
- осъществяват синхронизация между паралелните процеси в паралелния компютър.

5. Примери за КМ

Формално, всяка КМ се дефинира като набор от функции на вътрешните връзки. Всяка функция представлява взаимно еднозначно съответствие за целите числа от 0 до N-1 върху наборите входни/изходни адреси. Когато функцията f е приложена, данните от вход i се прехвърлят на изхода j ($j=f(i)$, $0 \le i \le N-1$). Понеже превключващата функция е взаимно еднозначно съответствие от набора цели 0,1,2,…….N-1 върху същия набор, то превключващата функция се явява пермутация.

5.1. КМ ILLIAC IV

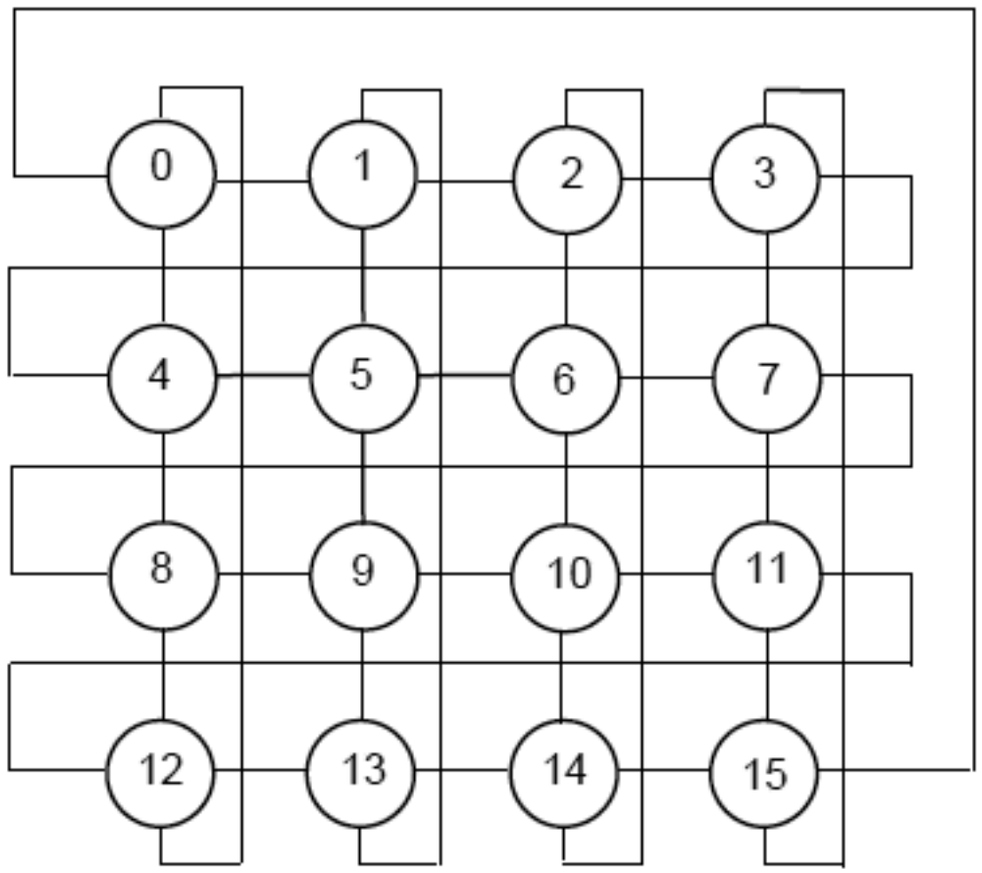
Тази мрежа се използва в паралелния компютър ILLIAC IV и се описва от четирите функции:

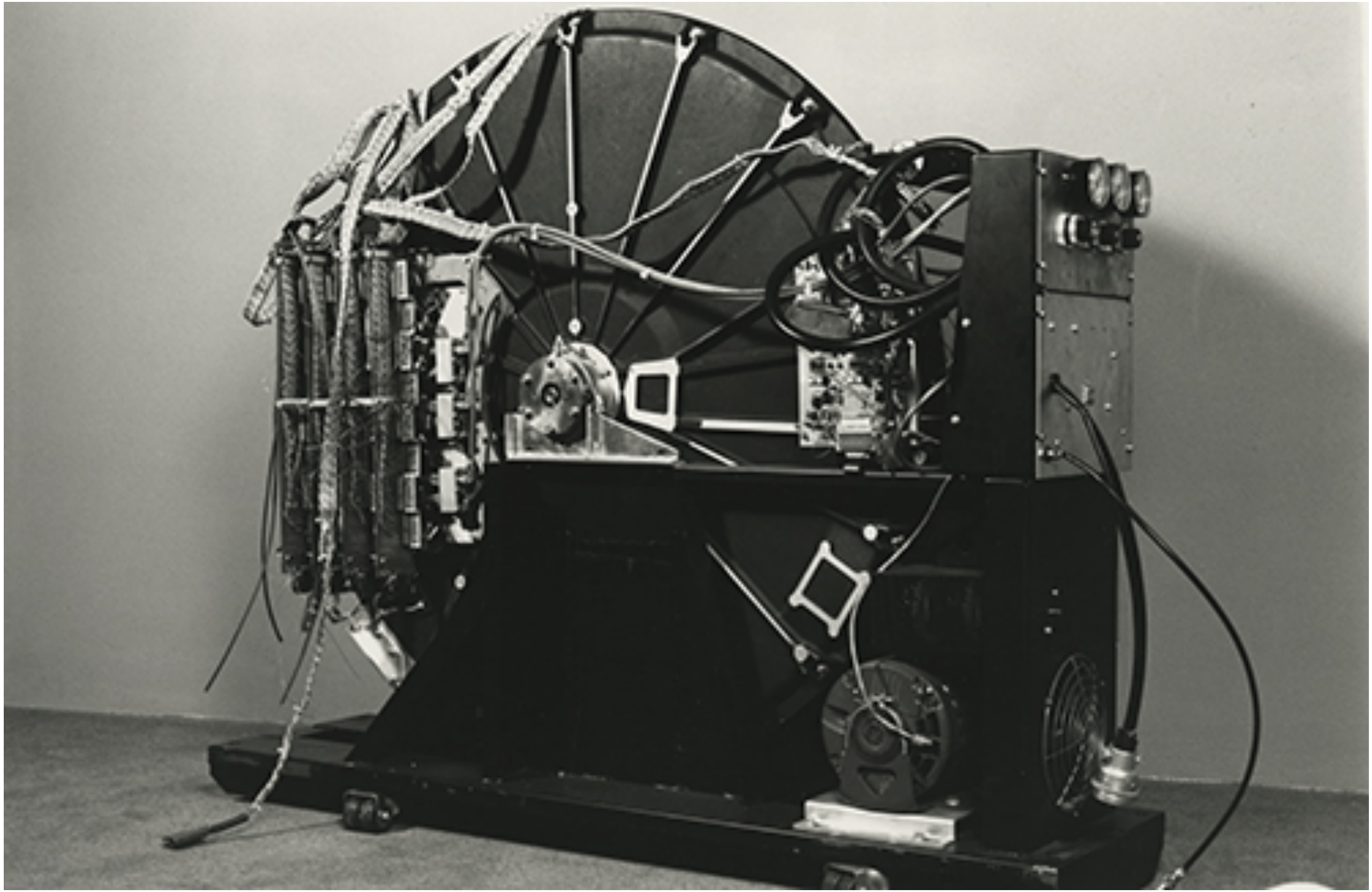
$$I_{+1}(i)=i+1 \bmod N$$
$$I_{-1}(i)=i-1 \bmod N$$
$$I_{+q}(i)=i+q \bmod N$$
$$I_{-q}(i)=i-q \bmod N$$

Където: $0 \le i \le N-1$, $\boldsymbol{q=\sqrt{N}}$;

и $i-x \bmod N=i+(N-x) \bmod N$, $x=\{+1,-1,+q,-q\}$.

На фиг.8-6 е показана такава мрежа за N=16.





ILLIAC IV represents a significant step forward in computer systems architecture offering

- greatly improved performances:
  - 200 MIPS computation speed
  - 10^9 bits/sec I/O transfer rate
  - 10^6 bytes of high-speed integrated circuit memories
  - 2.5×10^9 bits of parallel disk storage
- contemporary technology:
  - ECL circuits
  - semiconductor memories
  - belted cables
- and a new approach to the art of computing using parallelism, which offers an opportunity to programmers to utilize the vast power of the system as effectively as possible.

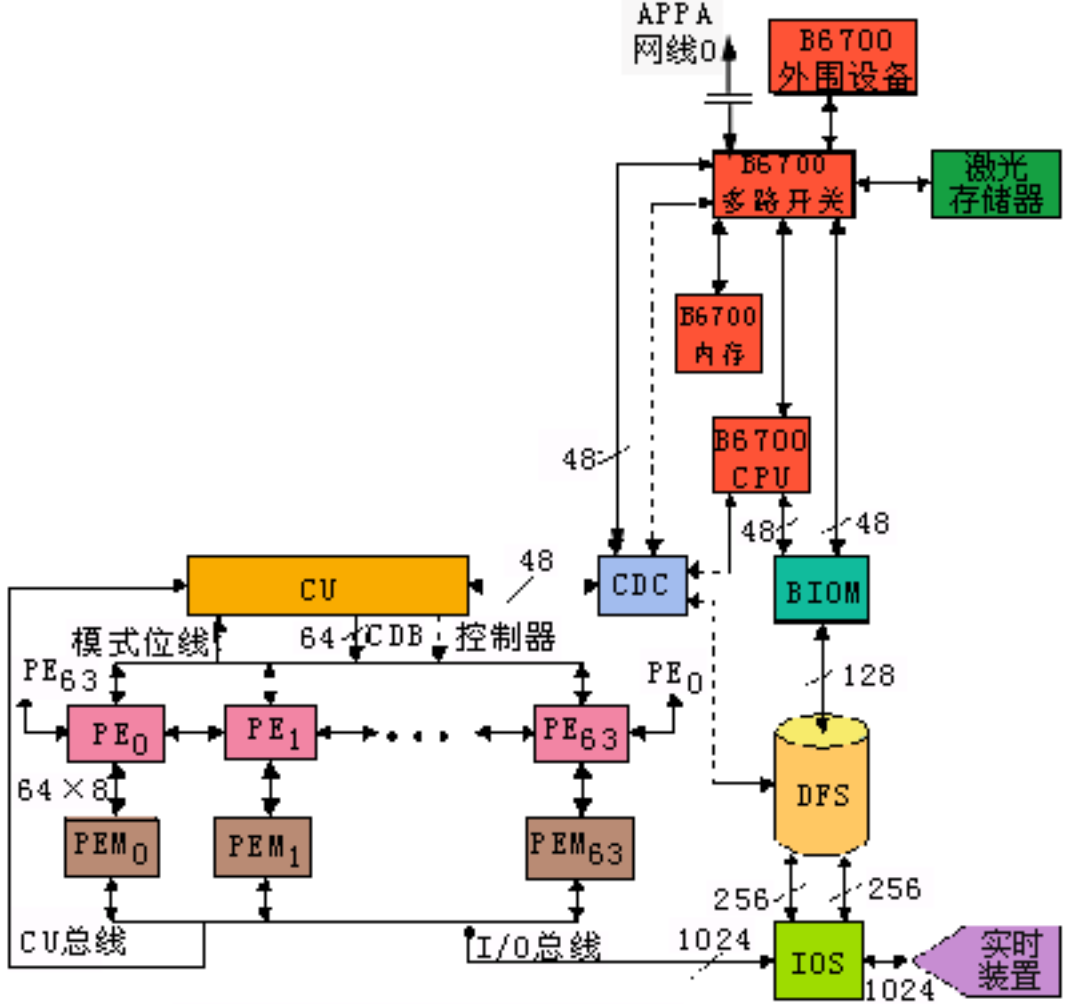


图10.6Illiac IV系统总框图

5.2. КМ от тип двоичен куб

Броят на входовете (процесорите) на обобщената хиперкубична мрежа се дава чрез съотношението

$$N=W^d,$$

където W е броят на върховете (процесорите) по всяка размерност на хиперкуба, а d е размерността на хиперкуба.

Очевидно, за получаване на големи стойности на N са възможни следните два случая:

а) W=const, а d расте;

б) W расте, а d=const.

При W=2 се получава мрежа двоичен хиперкуб. Тя е частен случай на обобщената хиперкубична мрежа и се състои от $d=\log_2 N$ превключващи функции от вида: $cube(q_{d-1}\ldots q_{i+1}q_iq_{i-1}\ldots q_0)=q_{d-1}\ldots q_{i+1}\bar{q}_iq_{i-1}\ldots q_0$ където $Q=q_{d-1}\ldots q_{i+1}q_iq_{i-1}\ldots q_0$, $0\le Q<N$, $0\le i<d$, а $\bar{q}_i$ е допълнението на q_i. Например, за N=8, $cube_0(0)=1$, $cube_1(0)=2$ и $cube_2(0)=4$.

Свойства на хиперкубичната мрежа. Двоичният хиперкуб Q_d съдържа $N=2^d$ възела и $d*2^{d-1}$ ребра. Ето защо при увеличаване на броя възли, броят на връзките за всеки възел расте пропорционално на d. Това определя тези практически ограничения на броя връзки на един възел, които да се отчитат при увеличаване на размерността на куба.

Всеки възел в куба се намира на разстояние единица от други d възли. Максималното междувъзлово разстояние, т.е. диаметъра на хиперкуба е равно на d и определя най-голямата задръжка при предаване на съобщения. Средното разстояние е $(d*2^{d-1}-1)/(2^d-1)$ и с увеличаване на d бързо се доближава до d/2.

Графа на d мерния двоичен хиперкуб може рекурсивно да се определи по следния начин:

$$Q_1 = K_2$$
$$Q_d = K_2 \times Q_{d-1}$$

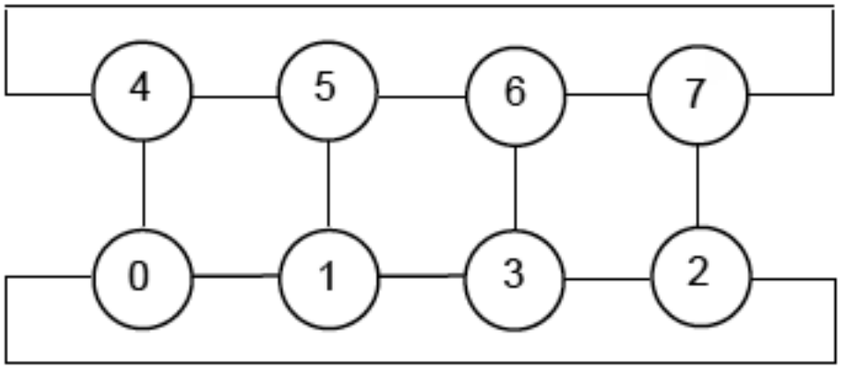
където K_2 двувърхов пълен граф, а x означава декартово произведение на графа. От това определение следва, че Q_d съдържа много подкубове с по-малка размерност. Това свойство може да бъде използвано по различен начин. Така например, на компютър с хиперкубична КМ могат да работят едновременно няколко клиента, като за всеки от тях се отдели подкуб. Така по друг начин могат да се решат въпросите за мултипрограмната работа, за разпределение и защита на паметта и др.

Хиперкубът притежава много привлекателни и полезни свойства. Например, той има регулярна структура, т.е. той се явява еднороден в смисъл, че структурата на системата изглежда еднаква от всеки възел. В понятията от теорията на графите, хиперкуба Q_d се явява симетричен, а това означава, че всяка двойка възли или линии може да бъде разместена без да се нарушава структурата на графа. Това свойство, съвместно с факта, че Q_d съдържа много лесно идентифицирани подкубове с размер по-малък от d, води до следните заключения:

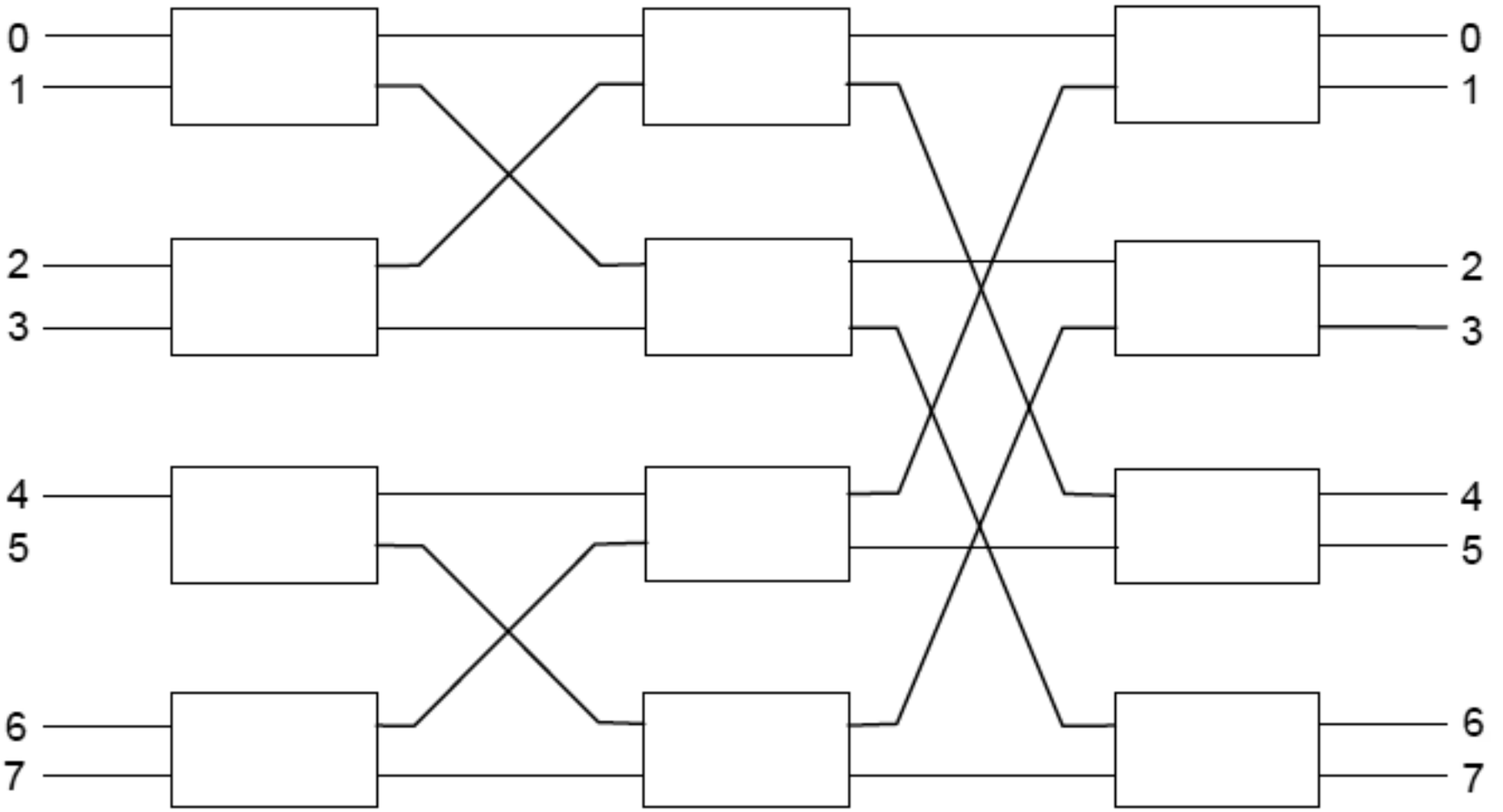
а) Програмата лесно може да бъде построена така, че тя да се изпълнява без изменение на хиперкуб с произволен размер. Това дава възможност програмата лесно да може бъде тествана и настроена на хиперкуб с по малка размерност (напр. две), а след това да се изпълнява на куб с по-голяма размерност.

б) Хиперкубът с по-голяма размерност може да бъде лесно използван от много клиенти едновременно, на всеки от които операционната система назначава подкуб.

Друго полезно свойство е възможността му в него да бъдат вложени други изчислителни структури, напр. дърво, кръг, решетка. Този резултат е особено важен защото дава възможност да се избере най-подходящата за дадена задача топология, така че да намалее до минимум прехвърлянето на данни между несъседни възли.



а) едностъпална мрежа от тип двоичен куб



б) многостъпална мрежа от тип двоичен куб при N=8.

5.3. Мрежа от тип shuffle-exchange

Тази мрежа се състои от две функции – на разместване (**shuffle**) и обмен (**exchange**). Функцията **shuffle** се дефинира като:

$$\textbf{shuffle}(q_{d-1}\ldots\ldots q_1 q_0) = q_{d-2}\ldots\ldots q_1 q_0 q_{d-1},$$

където $Q=q_{d-1}\ldots\ldots q_1 q_0$, $0 \le Q < N$ и $d=\log_2 N$.

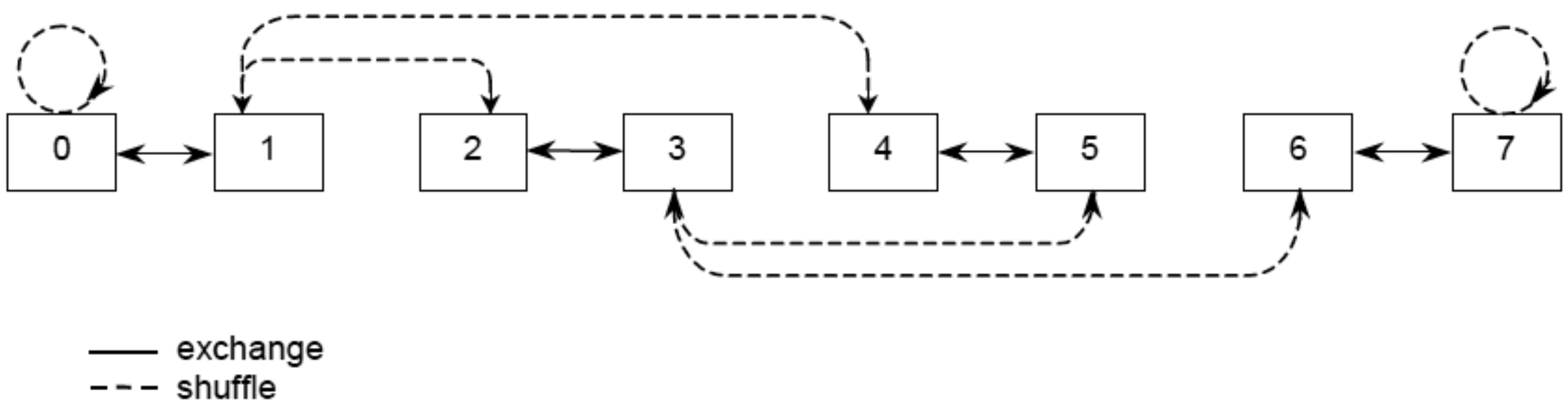
Например, за $N \ge 4$, shuffle(1)=2.

Функцията **exchange** се дефинира като:

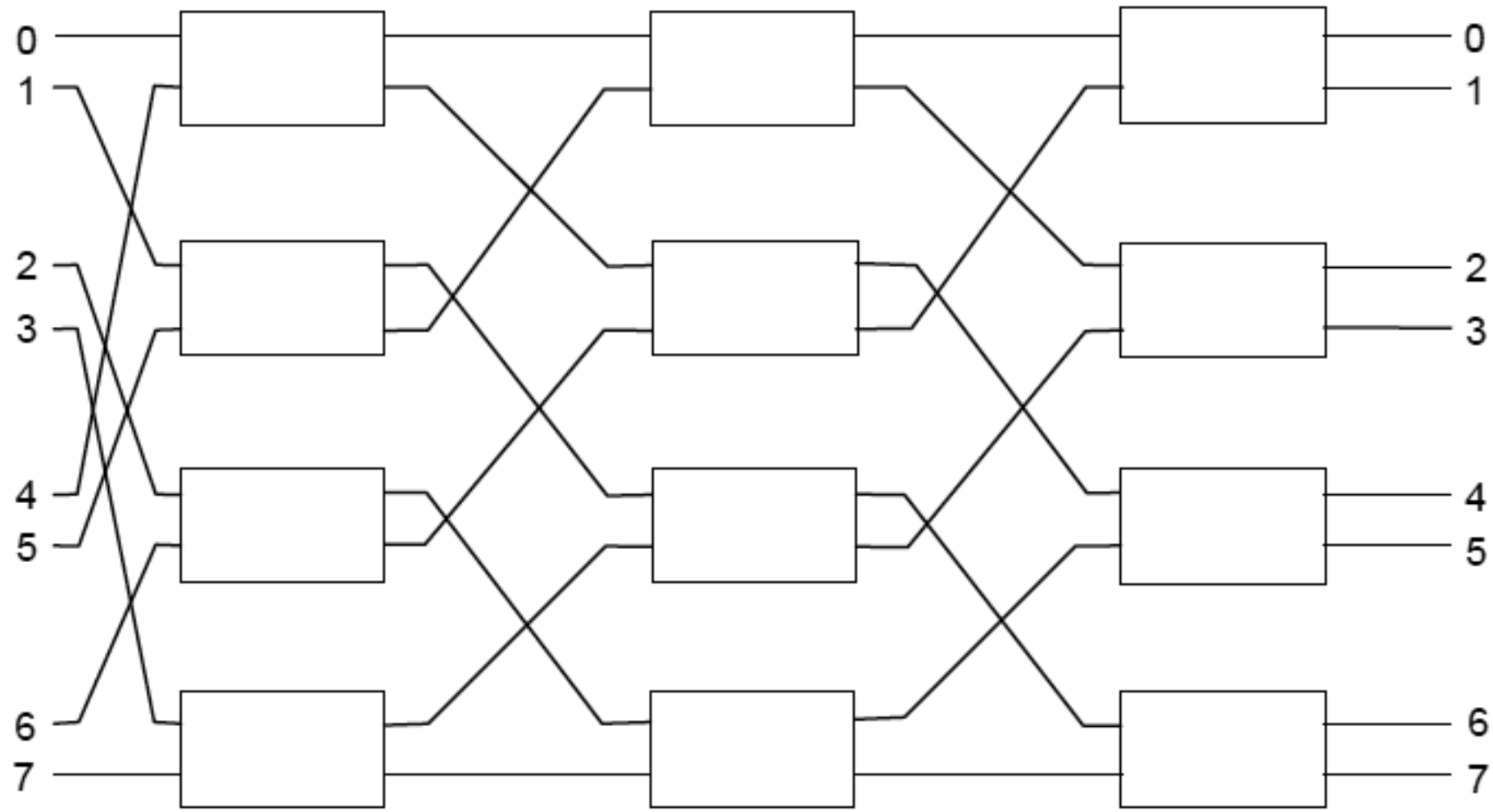
$$\textbf{exchange}(Q)=\mathrm{cube}_0(Q),$$

където $0 \le Q < N$. Например, за $N \ge 2$, exchange(1)=0.

Функцията **shuffle-exchange** също може да бъде приложена като едностъпална (рециркулираща) – фиг.12-9а или като многостъпална – фиг.12-9б мрежи.



а) Едностъпална мрежа от тип shuffle-exchange.



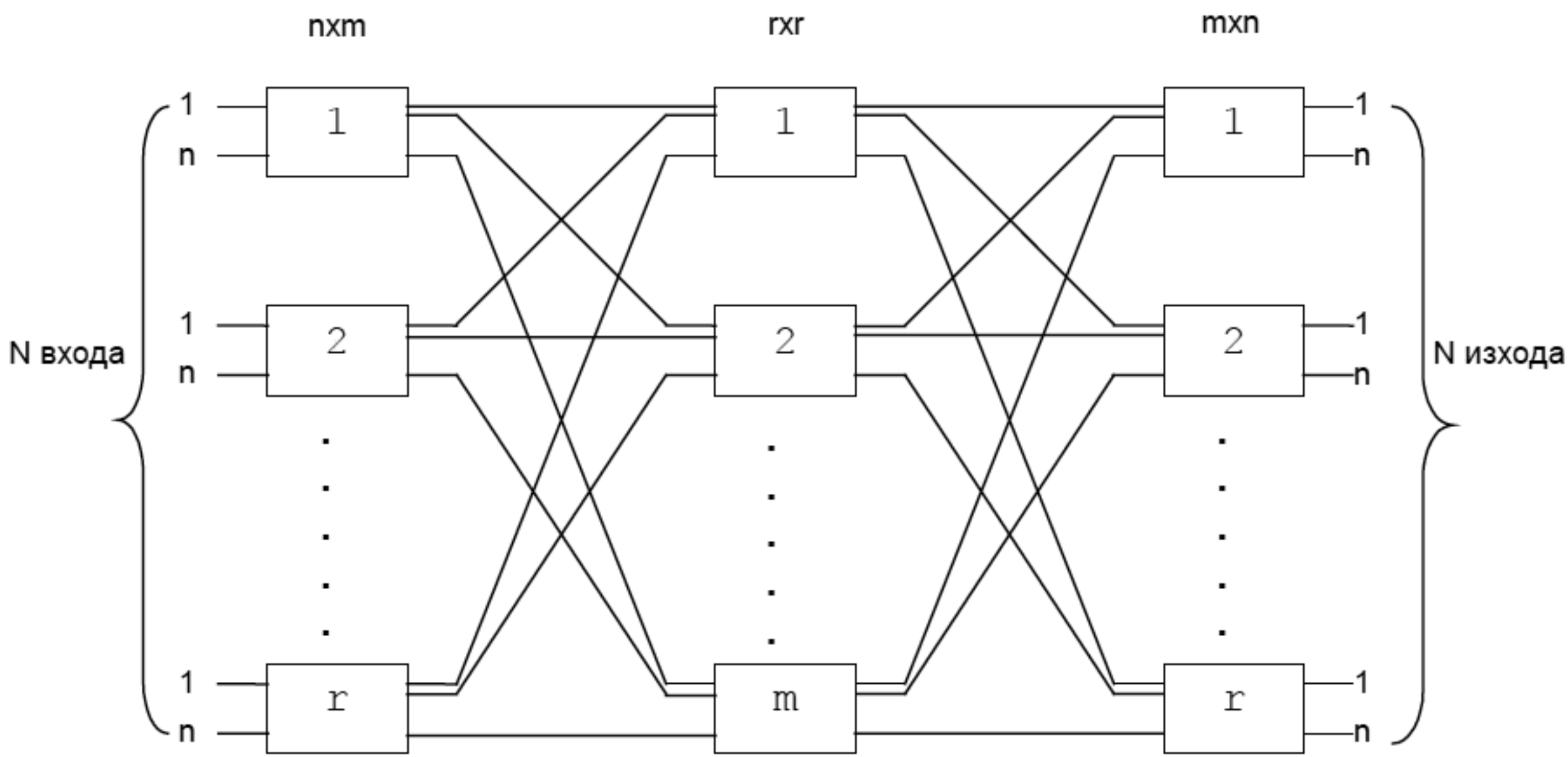
б) Многостъпална мрежа от тип shuffle-exchange.

Фиг.8-8. Примери за мрежи от тип shuffle-exchange.

Многостъпална мрежа от тип shuffle-exchange, както и многостъпалната мрежа двоичен куб, се състои от d стъпала. Всяко стъпало от shuffle-exchange мрежата осигурява shuffle връзките (свързване на линия с позиция x към позиция shuffle(x), $0 \le x < N$). Правилото за превключване на всеки КЕ и позициите, които той заема са същите както и за двоичния куб.

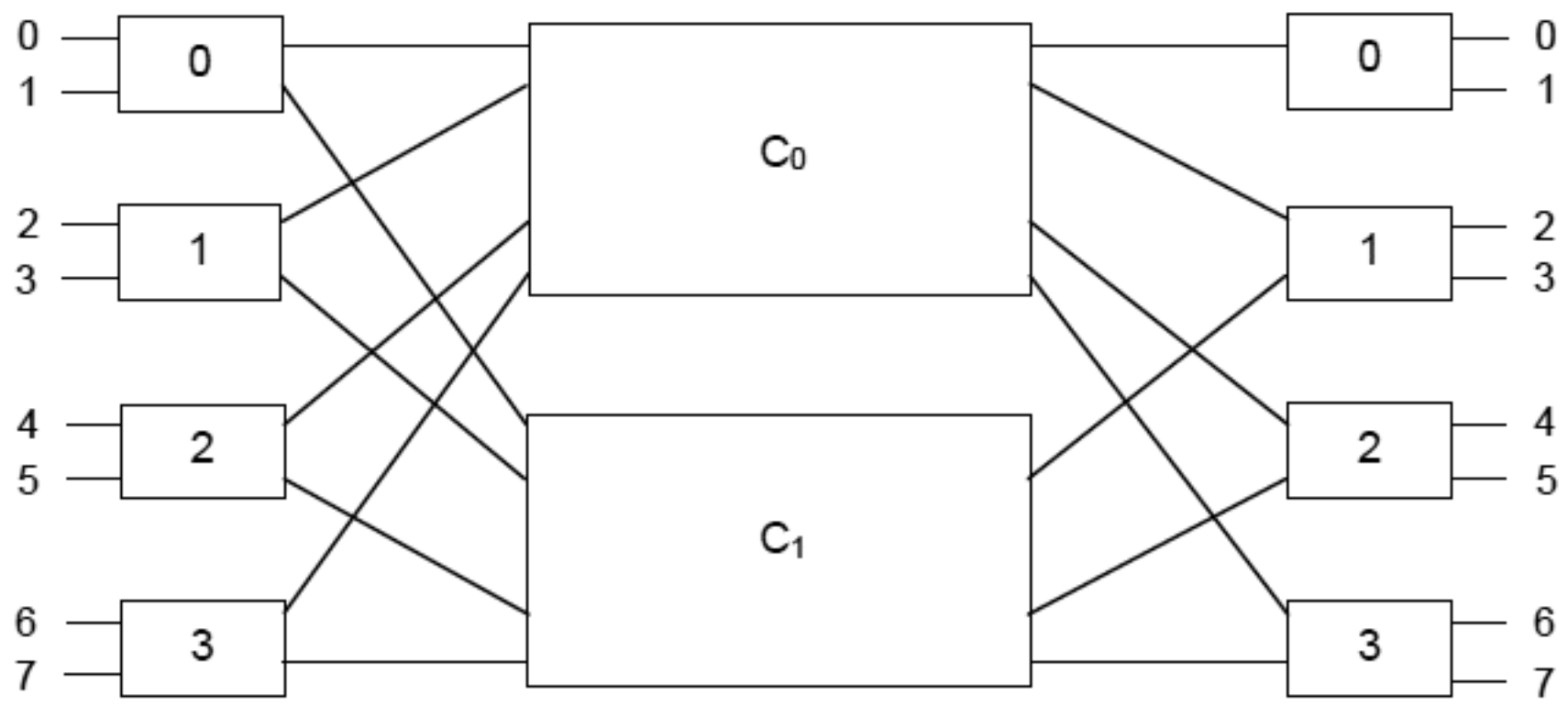
5.3. Мрежа на Клос.

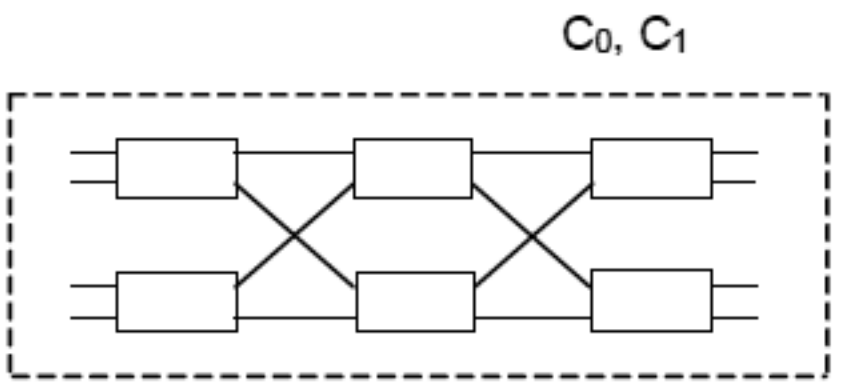
Мрежата на Клос има структура показана на фиг.12.10. КЕ тука са превключватели от типа nxm, rxr и mxn, където N=nxr. Ако е изпълнено условието m≥2n-1 мрежата е неблокираща. Частният случай на тази мрежа при m=n=r се нарича мрежа "Мемфис".



5.4. Мрежа на Бенес

Мрежата на Бенес с N входа и N изхода има симетрична структура - фиг.8-10, във всяка половинка на която между N/2 входа и N/2 изхода на КЕ е разположена също такава мрежа на Бенес, но с N/2 входа и N/2 изхода. КЕ имат две работни състояния $П_3$ и $П_6$; едната линия на КЕ се свързва към горната част на мрежата на Бенес, а другата - към долната част.Броят на стъпалата е равен на $2\log_2N-1$, а общия брой на КЕ в мрежата е $N(2\log_2N-1)/2$.





Тази мрежа се отнася към неблокиращите мрежи с реконфикурация, но поради факта, че за установяването на КЕ е необходимо време $O(N\log_2 N)$ динамичното съединение е невъзможно.

Фиг.9-1. Структура на паметта

За да се осигури висока ефективност на йерархично организираната памет е необходимо времената на достъп между различните нива да се различават на порядък [4]. Ето защо в съвременните компютри се въвежда масово кеш памет от второ ниво (напр. в P4 на Intel, G4 и G5 на Motorola, Duron на AMD и т.н.) и дори от трето ниво (в архитектурата Itanium-2 на Intel, K7, Athlon и Hammer x86-64 на AMD), а също така и дискова кеш памет към външната памет(виж тема 14).

В табл.9-1 са обобщени характеристиките на паметите по отделните нива.

Таблица 9-1

Ниво на паметта	Време за достъп	Типичен размер	Технология	Управление
Регистри	1-3 ns	1 KB	CMOS По поръчка(специална изработка)	От компилатора
L1 кеш (в чипа)	2-8 ns	8-128 KB	SRAM	Апаратно
L2 кеш (извън чипа)	5-12 ns	0.5-8 MB	SRAM	Апаратно
Основна памет	10-60 ns	61 MB-1 GB	DRAM	Операционна система
Твърд диск	3.10^6-10.10^6 ns	20-100 GB	Магнитна	Операционна система/Потребител

Максималната скорост на предаване на информацията от/към паметта е известна като ширина на лентата на предаване. За да може средната скорост на изчисление да не зависи от по-малката ширина на лентата на предаване от по-ниските нива, е необходимо така да бъде организиран изчислителния процес, че да е възможно най-голям брой команди и данни да се изпълняват чрез паметта от най-високо ниво, преди да възникне необходимост от презареждане от по-ниските нива. Прехвърлянето на данни (команди) между различните нива на паметта се извършва динамично, т.е. по време на изпълнение на програмата (програмите). Механизмът, който осъществява този процес се нарича виртуална организация на паметта.

Вертикалната структура на паметта води до:

- Подобрява работните характеристики на процесора.
- Увеличава пропускателната способност на компютъра.
- Осигурява достъп до големи количества данни за приемливо малко време.

1.2. Архитектура на кеш паметта

1.2.1 Работа на кеш паметта

Кеш паметта е ключа за осигуряване на производителност при съвременните процесори. Типично в една програма около 25% от инструкциите се отнасят до паметта, така че времето за достъп до паметта се явява критичен фактор в изпълнението на програмата. Чрез ефективно редуциране времето за достъп до паметта, кеш паметта позволява повече от една инструкция за цикъл да се изпълнява в модерните процесори (виж тема 5). Допълнителна индикация за важността на кеш паметта е напр., че от 6.6 10^6 транзистора в процесора MIPS R10000, 4.4 10^6 се използват в първичния кеш (в това число и TLB).

Работата на кеш паметта се базира на <u>локалността на обръщенията.</u>

Всички програми показват локалност на обръщенията. Това се оказва универсално свойство на програмите - независимо дали са комерсиални, научни, игри т.н. Кеш паметта експлоатира това свойство, подобрявайки времето за достъп и намалява цената на достъп до главната памет. Има два типа локалност

А) Локалност по време.

Б) Локалност по пространство (специална локалност).

Локалност по време - Веднъж направено обръщение към дадена позиция (клетка) от паметта има голяма вероятност, че тя ще бъде потърсена отново в недалечно бъдеще.

Примери: Най-простият пример за локалност по време е изпълнението на цикъл: веднъж въведен (включен) цикълът, всички команди в цикъла ще се посочват отново, вероятно много пъти, преди цикълът да завърши. Обобщено извикването на подпрограми, функции и прекъсванията по време също проявяват такова свойство.

Много типове данни показват локалност по време: в някаква точка на програмата съществува тенденция за обръщение към "горещи" данни, които програмата използва или променя многократно, преди да премине към друг блок от данни. Някои примери са:

- броячи;
- преглед на таблици;
- натрупване на променливи;
- стекови променливи.

Локалност по пространство - Когато се адресира клетка от паметта, много вероятно е, че съседните клетки ще бъдат скоро достъпни.

Примери: Ясно е, че потокът от инструкции ще проявява значителна специална локалност. В отсъствие на преходи следващата команда да бъде изпълнена е едно непосредствено, пряко следствие на текущата.

Данните също показват значителна специална локалност – напр. когато матрица или низ са достъпни, обработката започва от началото на матрицата до края, последователно.

Локалността по време включва локалността по пространство, но не и обратно.

Като правило, приложенията проявяват изключителна локалност по време за код и бедна за данни. Този факт е довел до разделянето на кеш паметта на две части – едната за команди (i-cache), а другата за данни (d-cache). Това разделяне води до значително увеличаване на производителността, зависещо от размера на кеш паметта, вида на приложението, др. Фактори.

1.2.3. Операции с кеш паметта

Един от основните въпроси при работа с кеш паметта е: "Как процесорът разбира дали информацията, която му е необходима, е налична в кеш паметта?"

Оперативната памет и кеш паметта са разделени на отделни области (frames), съдържащи определено количество байтове. Обменът на информацията между оперативната памет и кеш паметта се реализира по блокове. Размерът на блока съвпада с размера на фрейма и освен това в една област (един фрейм) може да се съхранява един блок. Когато процесорът отправи заявка към конкретен байт от определен блок в паметта, трябва много бързо да се определят следните три неща:

- дали търсения блок се намира в кеш паметта, т.е. дали има попадение в кеш паметта (cache hit) или не (cache miss);
- позицията на блока в кеш паметта в случай на наличие на блока в кеш паметта;
- положението на търсения байт в блока, отново само когато става дума за наличие на блока в кеш паметта.

Кеш памет с пълна асоциативност

При пълната асоциативност всеки блок от оперативната памет може да бъде свързан към всеки фрейм от кеш паметта. При това обаче се налага да се претърсва цялата tag RAM, за да бъде открит нужния блок. По тази причина колкото по-голяма е кеш паметта, толкова по-голямо ще е забавянето, преди нужния байт да стане достъпен за процесора. Ето защо пълната асоциативност не е често използвана организация в съвременните процесори.

Кеш памет с директно съответствие

При еш памет с директно съответствие на всеки блоков фрейм се съпоставя определено подмножество от основната памет. Например, ако разполагаме с 8 фрейма, във всеки от тях ще се разполага всеки осми блок от паметта, т.е. във фрейм 0 ще се разполагат блокове 0, 8, 16 и т.н., във фрейм 1 - блокове 1, 9, 17 и т.н. Голямото предимство в случая е, че потенциалните положения на всеки блок са значително редуцирани и съответно броят на търсенията в tag RAM е много по-малък. Например, ако на процесорът в нужен блок 0, 8 или 16 той трябва да провери само фрейм 0. Недостатъкът, който се появява обаче е следния - напр. на процесорът са нужни блокове с номера от 0 до 3 и от 8 до 11. При това положение двете групи блокове не могат да се поместят едновременно във фреймите, понеже 0 и 8 се разполагат във фрейм 0, а 1 и 9 - във фрейм 1 и т.н. При това положение процесорът ще трябва постоянно да променя съдържанието на кеш паметта, да отправя заявки към оперативната памет и като краен резултат рязко се снижава ефективността от кеширането и се губи преимуществото на намалената латетност. Такава ситуация се нарича колизия.

n-кратна (частична) асоциативност

Третият възможен вариант на организация представлява комбинация от горните два. При него определени подмножества от паметта могат да се поместят в определени групи фреймове. Например, четните блокове се поместват в първите 4 фрейма, а нечетните - във вторите 4. По този начин закъсненията при търсене в спомагателната памет се намалява два пъти спрямо пълната асоциативност, но все пак остава по-голямо отколкото при директното свързване и освен това вероятността да се достигне до колизия се намалява значително в сравнение с директното свързване. Когато процесорът определи в коя група би следвало да се намира информацията, той ще трябва да претърси само четири фрейма. Конкретно тази реализация се нарича четирикратно асоциативна кеш памет (four way set associative), поради факта, че кеш паметта е разделена на групи от по четири фрейма. Могат да се използват различни модификации на тази техника от типа двукратна (групи от по два фрейма) или осемкратна (групи от по осем фрейма) асоциативност. Не трябва да се забравя обаче, че всяко увеличаване на нивото на асоциативност (от две, към четири или от четири към осем) също увеличава броя на тяговете (етикетите), които трябва да се проверяват за да се определи конкретен блок. Следователно увеличаването на асоциативността също означава увеличаване на латентността.

1.2.4. Заместване на информацията в кеш паметта.

Важен въпрос при кеширането е кой точно блок от кеш паметта да бъде заместен, когато постъпва нов блок. Най-простите стратегии са: да се замести случаен блок или да се използва някоя от методиките LIFO (Last In, Last Out) или FIFO (First In, First Out). За съжаление никоя от горепосочените методиките не отчита локалността на обръщенията. По тази причина, по-добра стратегия на заместване е **LRU** (**L**east **R**ecently **U**sed - най-отдавна използван), т.е. на заместване подлежи блокът използван преди най-много време. Именно тази стратегия и намира най-голямо приложение в съвременните процесори.

1.2.5. Запис в кеш паметта

Досега разгледахме поведението на кеш паметта при операция четене. При операция запис, която се среща основно при работа с кеш паметта за данни, има една особеност на които ще се спрем по-долу. При получаване на резултат, процесорът го записва в кеш паметта от най-високо ниво. Въпросът е как и кога този резултат се прехвърля в основната памет. Има две стратегии:

- Да се запише променената информация само в съответния й блок в кеш паметта от първо ниво и да се актуализира по другите нива едва когато дадения блок бъде заместен. Това е така наречената стратегия write back.
- Да се запише променената информация едновременно по всички нива на йерархията на паметта. Това е така наречената стратегия write through.

По отношение на производителността, първата стратегия е по-добра, защото относително по-рядко се налага да се прехвърля модифицираното съдържание на блока към останалите нива на паметта. От друга страна втората стратегия позволява по-лесно да се поддържа валидността на информацията в многопроцесорните системи и при системи с интензивни входно-изходни процеси.

1.2.6. Съгласуваност на данните в кеш паметите **[5]**

В паралелните компютри от тип **SMP** (виж тема 10), където множеството процесори имат кеш памети на самия чип и използват обща памет, възниква един допълнителен проблем, известен под името "съгласуване на данните" или "кохеретност на данните" в кеш паметите. Същността на проблема се проявява в следното. В този тип компютри има едно копие от операнда в основната памет и по едно копие от него във всяка кеш памет. Когато копието от операнда е променено, да кажем от процесор #1, то следва и другите копия да бъдат променени също. Съгласуването на данните в кеш паметите е дисциплина с която, промените в стойностите на операндите, направени от отделен процесор, да се разпространят навсякъде в системата по подходящ начин и да бъдат достъпни за използване от другите процесори. Това става като на хардуерно ниво се въвежда "подслушване" на системния интерфейс. Един от най-често използваните протоколи за осигуряване на кохеретност на данните е **MESI** протокола, където буквите от акронима определят четирите състояния, в които кешовата линия може да бъде:

- **M**odified – линията е била модифицирана; копието на паметта е невалидно;
- **E**xclusive – кеш паметта има само едно копие на данните; паметта е валидна;
- **S**hared – повече от една кеш памет съдържа копие от тази линия; копието на паметта е валидно;
- **I**nvalid – кешовата линия е невалидна

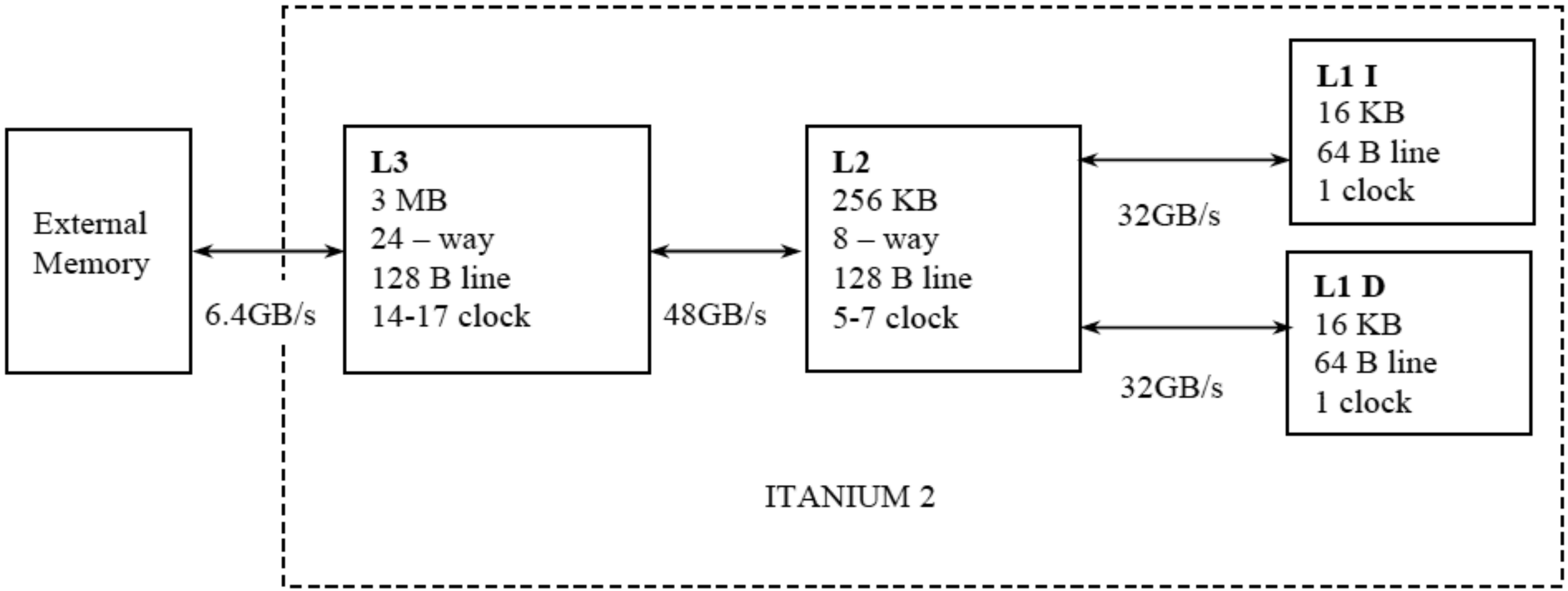
Латентността може да се дефинира като времето, необходимо на процесора да получи отговор на заявка за информация, която е отправил към паметта. Това включва:

- времето, нужно на заявката да стигне от процесора до паметта;
- времето на паметта да намери съответната информация;
- времето, за което данните достигат от паметта до процесора.

Всички тези времена зависят от конкретната организация на връзката процесор-памет. Обикновено латентността се дава с брой тактови цикли на процесора, които изминават между изпращането на заявката и получаването на отговора. Например, ако процесорът работи на 1 GHz, а тактовата честота на шината е 100MHz, отношението на тактовите честоти е 10:1, т.е. когато процесорът подаде заявка за четене към паметта, ще получи отговор най-рано след 20 цикъла (като не се счита времето на паметта да намери търсената информация), т.е. латентността е 20 процесорни цикъла. През това време процесорът стои и чака. Този резултат е валиден при условие, че ефективността на паметта и системната шина е 100%, което в реалния свят не е изпълнено.

Пропускателната способност се дефинира като скоростта, с която информацията стига до процесора и се измерва с байтове за секунда. По принцип, когато се говори за пропускателна способност, се цитира максималната пропускателна способност, която никога не може да се постигне. Това е така защото по шината непрекъснато трябва да се предава информация. Това не може да се получи по няколко причини. Първо, процесорът за да изпрати заявка до паметта, трябва да заеме шината. След като бъде изпратена заявката, паметта трябва да намери съответните данни и на свой ред да заеме шината и на края - да предаде данните. По тази причина натоварването на шината с предаването на данните никога не може да бъде 100%.

Пропускателната способност и латентността са свързани. Връзката между латентността и пропускателната способност е ефективната пропускателна способност. Тя представлява реално предаденото количеството информация. Например, ако паметта е способна да предаде данните на шината след 3 тактови цикъла, това означава, че латентността на шината е 3 цикъла. Това означава, че на 4-тия такт процесорът ще разполага с данните, т.е. ефективността е 25%. Ако се приеме, че честотата на шината е 133 MHz, а ширината й е 8 байта, то максималната пропускателна способност е 1064 MB/s (133.10^6 [Hz] x 8 [bytes]=1064MB/s). Но поради ефективност от 25%, то ефективната пропускателна способност е едва 266 MB/s. Една от използваните техники за намаляване на латентността на шината е освен изискания от процесора байт да се предадат и няколко съседни на него байта (локалност по пространство). Поради факта, че в кеш паметта трябва да се помести цял блок от системната памет, информацията обикновено се предава на цели блокове. Нека приемем, че размерът на един блок е 32 байта. Тогава след първото предаване от 8 байта ще последват още 3, но за тях няма да е необходимо изчакване от 3 цикъла. Така се получава 3 празни цикъла и 4 ефективни (пълни с данни) цикъла или ефективността на шината е $\frac{4}{7} \approx 57.1\%$ т.е ефективната пропускателна способност нараства до 607.5 MB/s. Този механизъм на предаване се нарича Burst Mode.



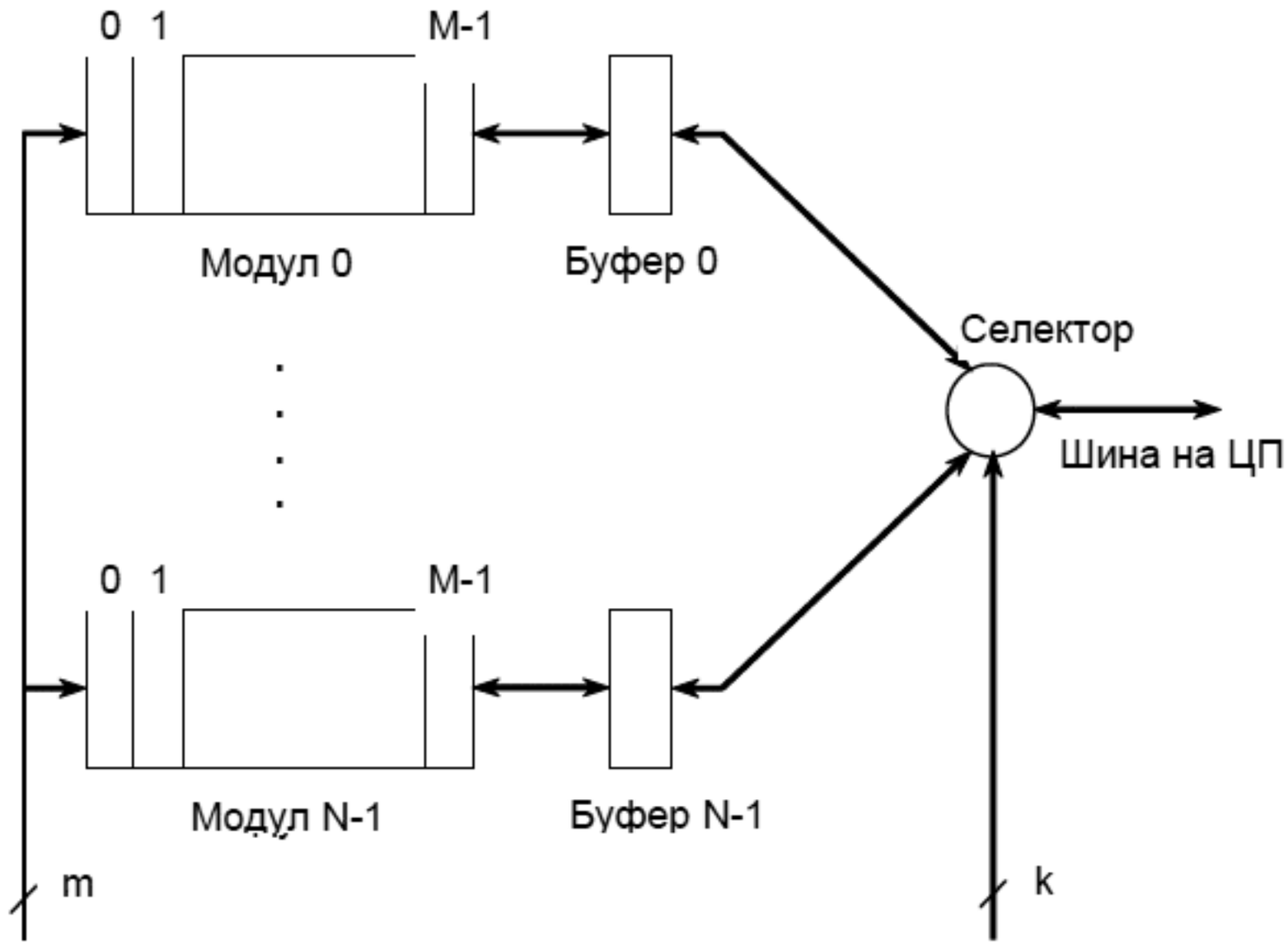
Фиг.9-2. Организация на кеш паметта в Itanium 2

Трите нива - L1, L2 и L3 на кеш паметта са разположени в чипа, заедно с ядрото на процесора. Прави впечатление, че паметта с по-ниско ниво е с по-малък обем от предходното, но времето за достъп е също по-малко. Освен това промяната засяга и големината на кешовата линия (блок), типа асоциативност, също така и скоростта на трансфер. Стратегията за запис е through write.

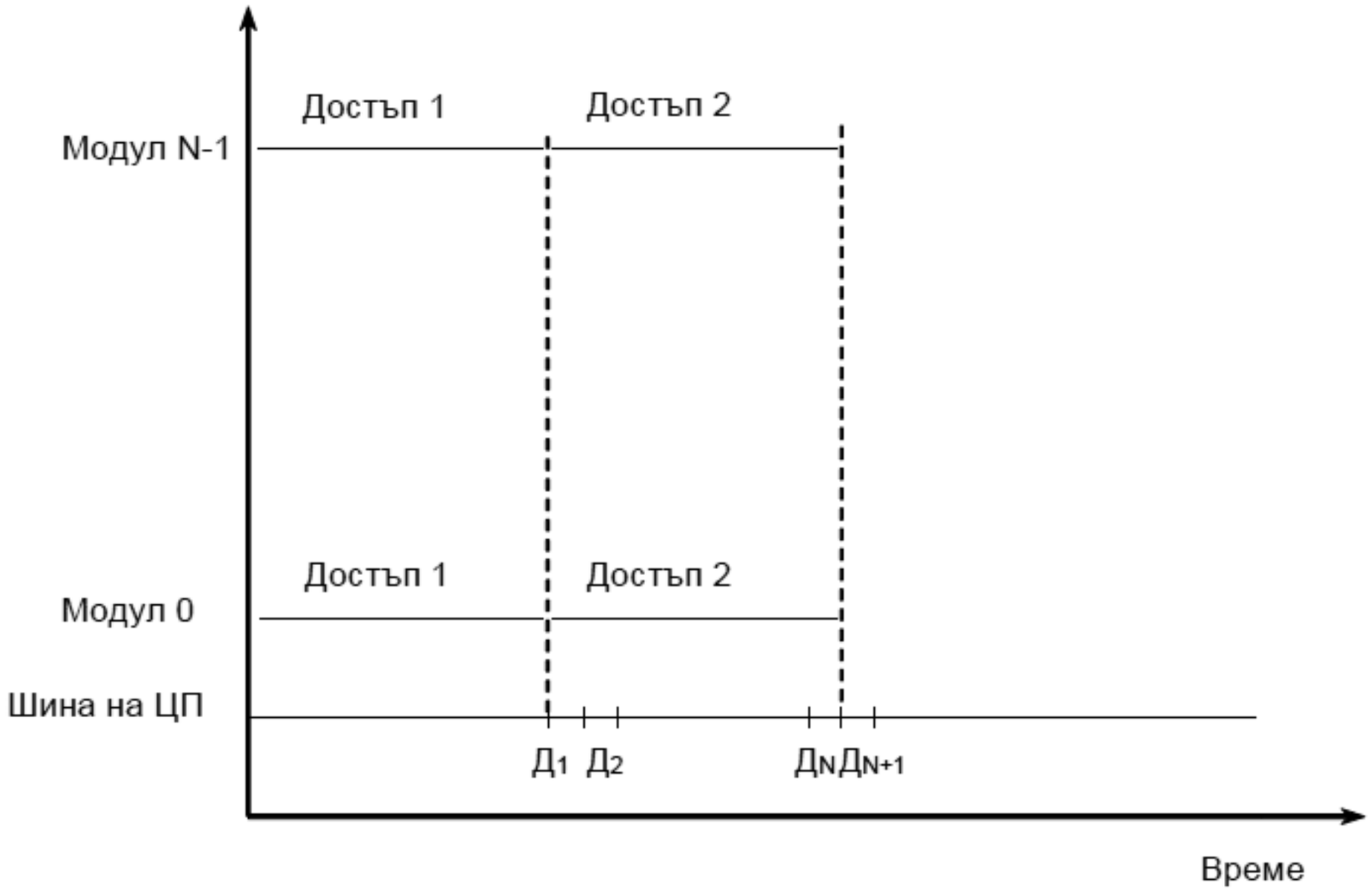
Хоризонтална организация на паметта може да бъде изградена по два начина:

- пакетна обработка на множество достъпи към паметта;
- конвейерна обработка на множество достъпи към паметта.

На фиг.9-3 е дадена примерна структура на памет от първия вид, осигуряваща достъп до **N** думи паралелно при всяко обръщение. Ако $\mathbf{N=2^k}$ и $\mathbf{M=2^m}$, то **m** разряда от **m+k** разрядния адрес се изпращат към всички модули памети, а младшите **k** разряда се използват за избор, коя от **N-те** думи да се чете първа. Така при всяко обръщение към паметта се прочитат **N** последователни думи (с адреси iN+j, където $0 \le j \le N-1$). Тези думи се поместват в буфери (фиксатори) и при следващото обръщение се четат от там със скорост **N** пъти по-голяма, отколкото времето за достъп до модула.

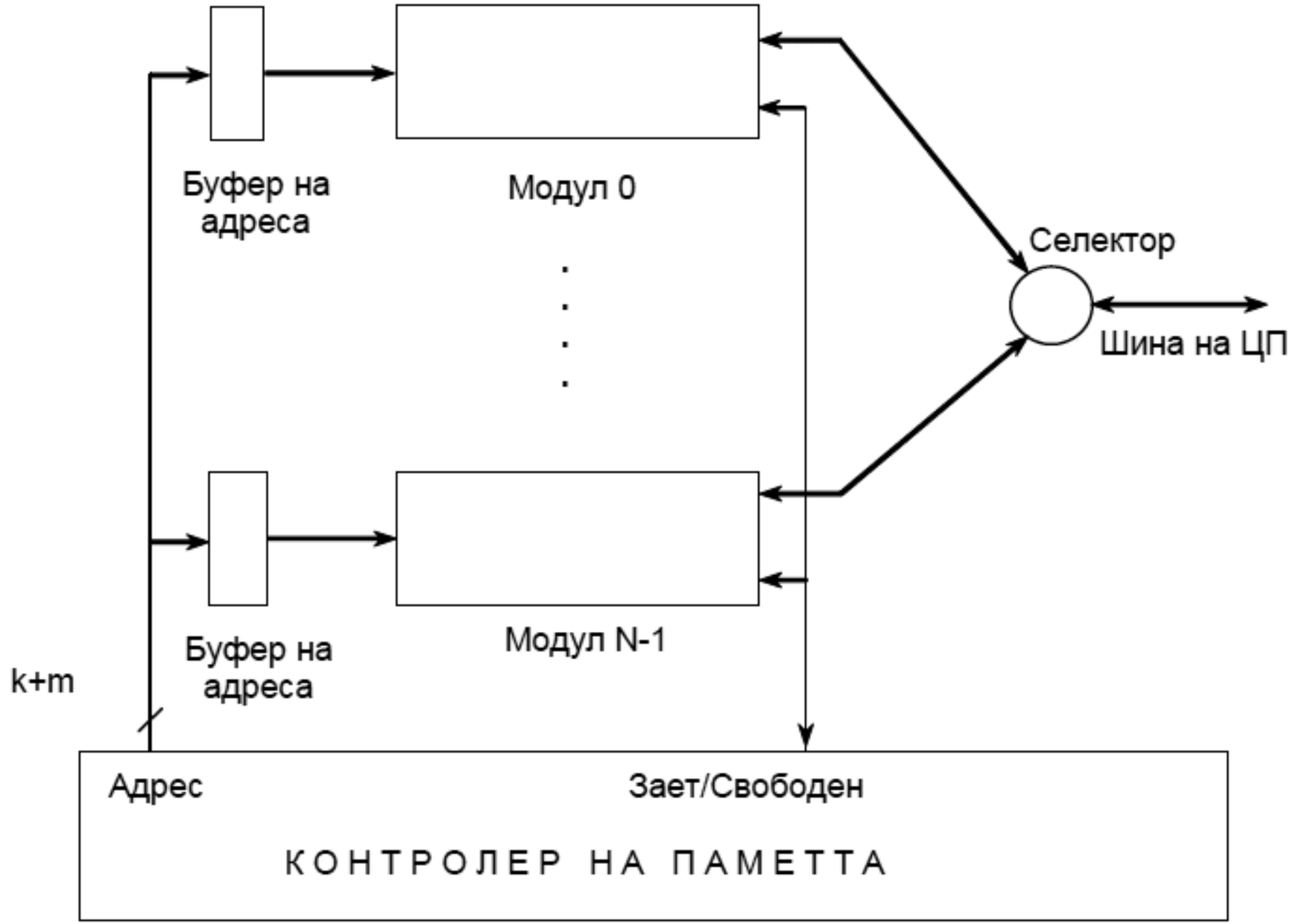


Фиг.9-3. Пакетен достъп до паметта с използуване на редуване на адресите

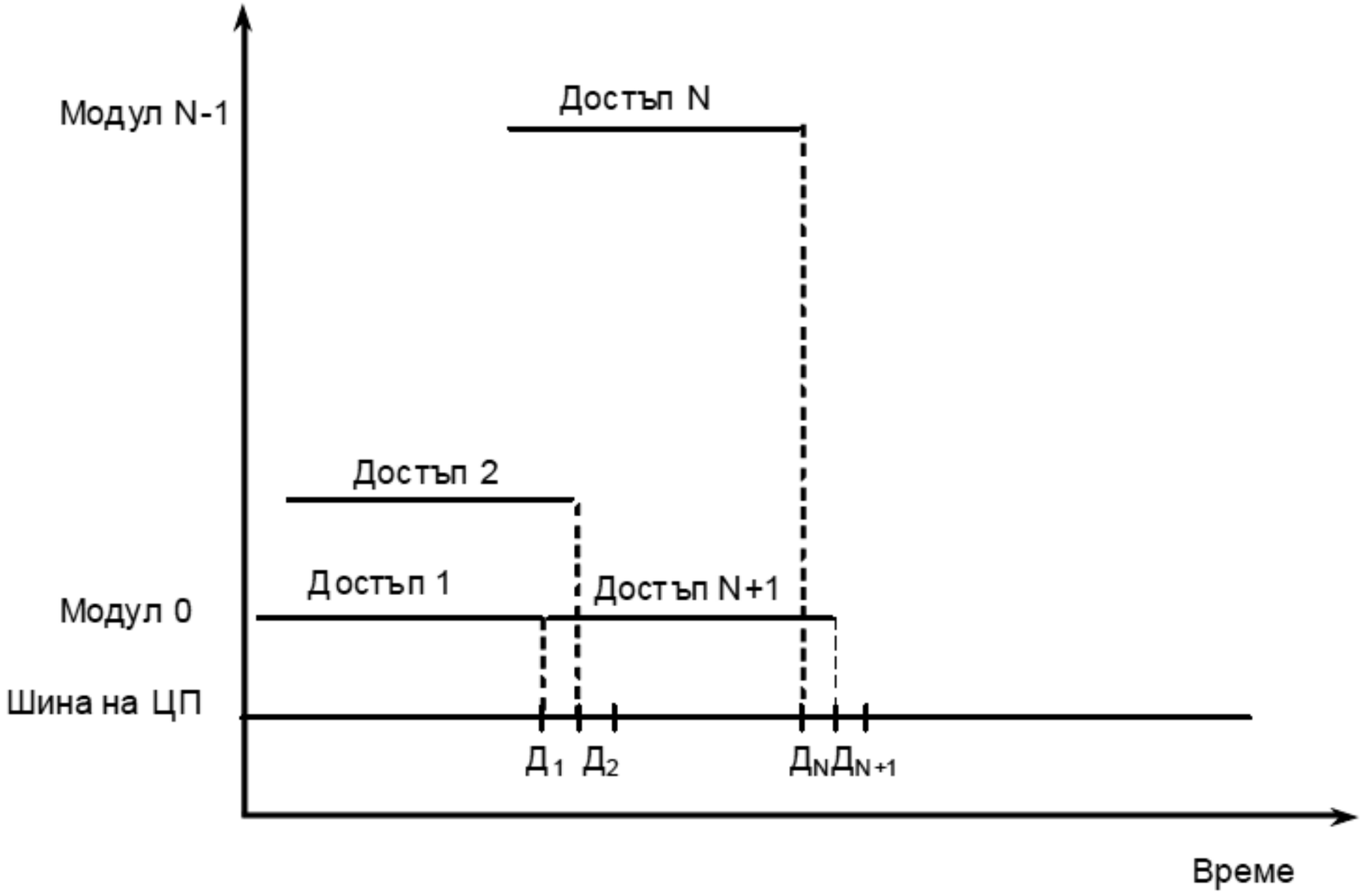


Фиг.9-4. Диаграма за четене на N думи в пакет

Тази проста структура се явява идеална в тези случаи, когато обръщението към паметта се осъществява последователно, както при избор на вектори във векторните процесори така и при изпълнение на последователни команди. Но времето за достъп нараства значително когато е нужен достъп до непоследователно разположени думи (например, в програми с висок процент на команди за преход). Може да се докаже, че за последователност с интервал **q**, средното време за достъп към елементите на паметта ще бъде **qt/N** при **q≤N** и **t** при **q>N** (**t** е времето за достъп до произволен модул).



Фиг.9-5. Разслоена памет с конвейерна обработка на заявките.



Фиг.9-6. Времедиаграма на работата на конвейерната памет.

Двете разгледани системи ще обработват заявките на последователните думи **N** пъти по-бързо, отколкото отделния модул. Освен това, те със същата скорост ще обработват произволна серия от заявки, в която нито един модул в **N** последователни заявки не се използува повече от един път. Например, 8-кратно разслоената памет от фиг.9-4 ще работи 8 пъти по бързо отколкото единия модул, ако последователността на адресите са разпределени равномерно не само с интервал 1, но и с интервал 3,5,7,9,11…. За интервал 3 последователността на адреси е 0,3,6,9,12,15,18….., а последователността на използуваните модули е 0,3,6,1,4,7,2,5,…. При някои други значения на интервала, например 2, работата няма да бъде с такава максимална скорост, но все пак по-бърза в сравнение с един модул. Може да се докаже, че в общия случай работата за произволна равномерна последователност от адреси с интервал, отличен от **N**, ще се осъществява на **N** кратно разслоена памет най-малко два пъти по-бързо, отколкото за памет имаща само един модул.

Както стана ясно, разслоената памет е ефективна при подходящо обръщение към модулите й. За да се осигури такова обръщение се използва съответно разположение на данните в паметта. За да стане ясно какъв е проблемът, да разгледаме съхраняването на матрица X(4,4) в четирикратно разслоена памет. На фиг.9-7 е показан един от възможните начини.

Модул	0	1	2	3
	x_{11}	x_{12}	x_{13}	x_{14}
	x_{21}	x_{22}	x_{23}	x_{24}
	x_{31}	x_{32}	x_{33}	x_{34}
	x_{41}	x_{42}	x_{43}	x_{44}

Фиг.9-7. Разположение на елементите на матрицата X по редове.

В този случай е възможно безконфликтно обръщение към елементите от един ред или от диагоналите, т.е. извличат се 4 елемента за един цикъл на паметта. При опит да се обработват паралелно елементите на матрицата принадлежащи на един стълб, настъпва конфликт, защото всичките те се съхраняват в един и същ модул памет. За решаването на този проблем се използва така нареченото скосено (skewed) разположение - фиг.9-8.

Модул	0	1	2	3
	x_{11}	x_{12}	x_{13}	x_{14}
	x_{24}	x_{21}	x_{22}	x_{23}
	x_{33}	x_{34}	x_{31}	x_{32}
	x_{42}	x_{43}	x_{44}	x_{41}

Фиг.9-8. Скосено разположение на елементите на матрицата X

В езика TRANQUIL за ILLIAC-IV са били предвидени средства, използувани от програмиста за указване способа на разместване в паметта. Операторът STRAIGHT подрежда данните по начин показан на фиг.9-7, а операторът SKEWED подрежда данните по начин показан на фиг.9-8.

Модул 0	1	2	3	4
x_{11}	x_{12}	x_{13}	x_{14}	
x_{24}		x_{21}	x_{22}	x_{23}
x_{32}	x_{33}	x_{34}		x_{31}
	x_{41}	x_{42}	x_{43}	x_{44}

Фиг.9-9. Напълно безконфликтен достъп до елементите на X.

Тук се използват повече на брой модули памет от броя процесори (в случая 5). Празните позиции в съответните клетки означава, че в тях не се записват елементи на матрицата X.

Може да се докаже, че за да се достигне степен на паралелизъм **N**, модулите памет **M** трябва да бъдат първото просто число по-голямо от броят на процесорите **N**. Така е решен проблемът в компютъра BSP на Burroughs. Броят на модулите памет е 521 (първото просто число по-голямо от 512 - броят на процесорите).

Обслужването на една заявка, направена към диска, зависи от следните три основни компоненти:

- време за достъп;
- време за изчакване;
- време за предаване на данните.

Време за достъп - това е времето необходимо за позициониране на главите на съответната пътечка, съдържаща търсените данни. То се явява функция от загубите по началното ускорение на главите, а така също и функция от броя на пътечките, които се пресичат по пътя към търсената пътечка.

Време за изчакване това е времето необходимо да се завърти диска и търсеният сектор да стане съвместим с положението на главата, след което данните могат да бъдат записвани или четени. При съвременните дискови устройства типичната скорост на въртене е 7200 об/мин, което дава пълен оборот на диска от порядъка на 8 мсек, а средното време за изчакване е от порядъка на 4 мсек. Разбира се има дискове, които се въртят и с по-висока скорост - 10000 об/мин и дори 15000 об/мин. Важно е да се отбележи, че в най-лошия случай времето за изчакване е сравнимо със средното време за достъп.

Време за предаване на данните - това е времето, необходимо за физическото предаване на байтовете информация между диска и компютъра. Това време се явява функция от броя предадени байтове.

За дадено работно натоварване съществуват три метрики, които могат да характеризират заявките за вход/изход:

- производителност;
- време на изчакване;
- пропускателна способност.

Производителността се определя като брой заявки за обслужване, получавани за единица време.

Времето на изчакване определя времето, необходимо за обслужване на индивидуална заявка.

Пропускателната способност определя количеството данни, предавани между устройствата, изискващи обслужване и устройствата, изпълняващи обслужване.

Предполагайки интензивността на отказите постоянна, т.е. при експоненциален закон на разпределение на появяване на отказите, а също така, че отказите са независими, то се получава, че средното време за безотказна работа (**M**ean **T**ime **T**o **F**ailure - **MTTF**) е:

$$MTTF_{\text{на дисковата матрицата}} = \frac{MTTF_{\text{на един диск}}}{\text{брой дискове в матрицата}}$$

За повишаване на устойчивостта на откази, се налага да се жертва пропускателната способност на вход/изхода или капацитета на паметта. Трябва да се използват допълнителни дискове, съдържащи излишна информация, позволяваща да се възстановят изходните данни в случай на отказ на диск. От тука се получава абревиатурата **RAID** (**R**edundant **A**rray of **I**nexpensive **D**iscs - матрица от евтини дискове с излишък). Едно по-съвременно дешифриране на тази абревиатура е **R**edundant **A**rray of **I**ndependent **D**iscs - матрица от независими дискове с излишък.

Когато дискът откаже, се предполага, че в продължение на кратък интервал от време отказалият диск ще бъде заменен и информацията ще бъде възстановена на новия диск с използването на резервната информация. Това време се нарича средно време за възстановяване (**M**ean **T**ime **T**o **R**epair – **MTTR**). Този показател може да се намали, ако в системата влизат допълнителни дискове в качеството на "горещ резерв"; при отказ на диска, резервният диск се включва. Четирите основни етапа на този процес се състоят в следното.

- определяне на отказалия диск;
- отстраняване на отказа без преустановяване на работата;
- възстановяване на загубените данни от резервния диск;
- периодична замяна на отказалите дискове с нови.

По-нататък ще се използват следните означения:

D – общ брой дискове в групата (D=C+G);

G – брой дискове с данни в групата;

C – брой проверяващи дискове в групата;

За **RAID** с изправяне на единични грешки без отчитане на отказите в спомагателното оборудване като захранващ източник, кабели и т.н., **MTTF** се дава чрез съотношението:

$$MTTF_{матица} = \frac{MTTF_{диск}^{2}}{(D + C + \frac{D}{G}) * (G + C - 1) * MTTR}$$

Както беше посочено в т.2, класовете високопроизводителни устройства за вход/изход имат тенденция да се различават по способа и по скоростта на достъп. Ето защо е полезно да се въведат различни метрики за тяхната оценка.

• При суперкомпютър метриката е брой записи или четения на големи масиви от данни за секунда (под голям масив се разбира масив съдържащ поне един сектор на всеки диск от групата). По време на предаването на големи масиви, всички дискове в устройството работят като едно устройство, при това всеки диск прави запис или четене на части от масива паралелно.

• При обработка на транзакции метриката е брой индивидуални (т.е на всеки отделен диск) записи или четения в секунда.

По този начин, в случай на суперкомпютри имаме работа с висока пропускателна способност при предаване, докато при обработка на транзакции ние говорим за висока скорост при вход/изход.

RAID 0*: *Разделяне на данните.

RAID 0 не е с излишък, следователно тази организация не се вписва напълно в акронима **"RAID"**. Тука данните са разпределени по битове или блокове върху множество от дискове. Възможностите за независимо четене или запис води до високо бързодействие. Понеже няма запомняне на излишна информация, производителността е много добра, но отказването на един диск в матрицата води до загуба на всичките данни.

RAID 1: Огледални дискове.

Петдесет процента от наличните дискове съхраняват допълнителна информация, т.е. G=C=D/2. Този способ се явява най-скъпият от разглежданите, защото всички дискове се дублират (G=1 и C=1) и при всеки запис се записва информация и на резервния диск. Познати са две разновидности на тази конфигурация:

• Един контролер за цялата матрица от дискове. Тъй като за всеки логически запис се извършват два физически записа, матрицата може да поддържа само половината от броя на физическите записи, възможни в случай на D независими диска.

• Два контролера – по един за дисковете с данни и за огледалната информация (резервните данни). Тази разновидност е позната като дублиране. Идеята може да се разпространи към останалите компоненти на дисковата подсистема – захранването и кабелите. Фирмата Tandem Computers прилага този способ в своите компютри с цел повишаване на устойчивостта на откази.

RAID 2: Разслоение по битове / Множество дискове за контрол.

RAID 2 е първото ниво използваща важната техника контрол по четност (**ECC E**rror **C**orrecting **C**ode). **RAID 2** е замислен за използване на дискове, които нямат вградено откриване на грешки и прилага коригиращите кодове на Хеминг. Решението, използвано в RAM паметите може да се повтори тука по пътя на побитово разслоение на данните и записът им на дисковата група, допълнена с достатъчно количество контролни дискове за откриване и коригиране на единичните грешки. Така групата при **RAID 2** се състои от G дискове с данните и C коригиращи дискове. Разрядите, записани върху всеки от отказалите дискове, могат да бъде възстановени с използване на метода на коригиращите кодове. За коригиране на единичните грешки и откриване на двойните грешки е необходимо $\lceil \log_2 G \rceil$ коригиращи диска т.е. контролните дискове съставляват $C = \lceil \log_2 10 \rceil = 4$ и дисковете с данни. Например, при G=10 диска,

сумарният	брой дискове е G+C=14.	Това дава 71% използваем
капацитет	на паметта (10:14=0,71)	и излишък на дискове 40%

(4:10=0,4). При G=25, $C = \lceil \log_2 25 \rceil = 5$ и сега общият брой дискове е равен на 30, а използваемия капацитет е 83%, като излишъкът от дискове е само 20%.

RAID 3: Разслоение по байтове / Един контролен диск.

Тука разделянето на данните е на ниво байтове и те са записани върху няколко диска (G на брой). Един диск е използван за съхранение на контролната информация, т.е. C=1. Така общият брой дискове е D=G+1. Намалявайки броят на контролните дискове до един на група се намалява излишната памет и ефективната производителност на отделен диск нараства, тъй като се изисква по малък брой контролни дискове. Освен това контролната информация се изчислява като се използва логическата функция **XOR**.

RAID 3 е толкова бърза колкото и **RAID 0** за операция четене, но записа върху единствения диск за корекции означава, че опашката за записи към диска бързо ще расте, дори и да се записват малки по обеми данни. Следователно тази конфигурация е по-подходяща за суперкомпютрите, отколкото за компютри обработващи транзакции.

RAID 4: Разслоение по блокове / Един контролен диск.

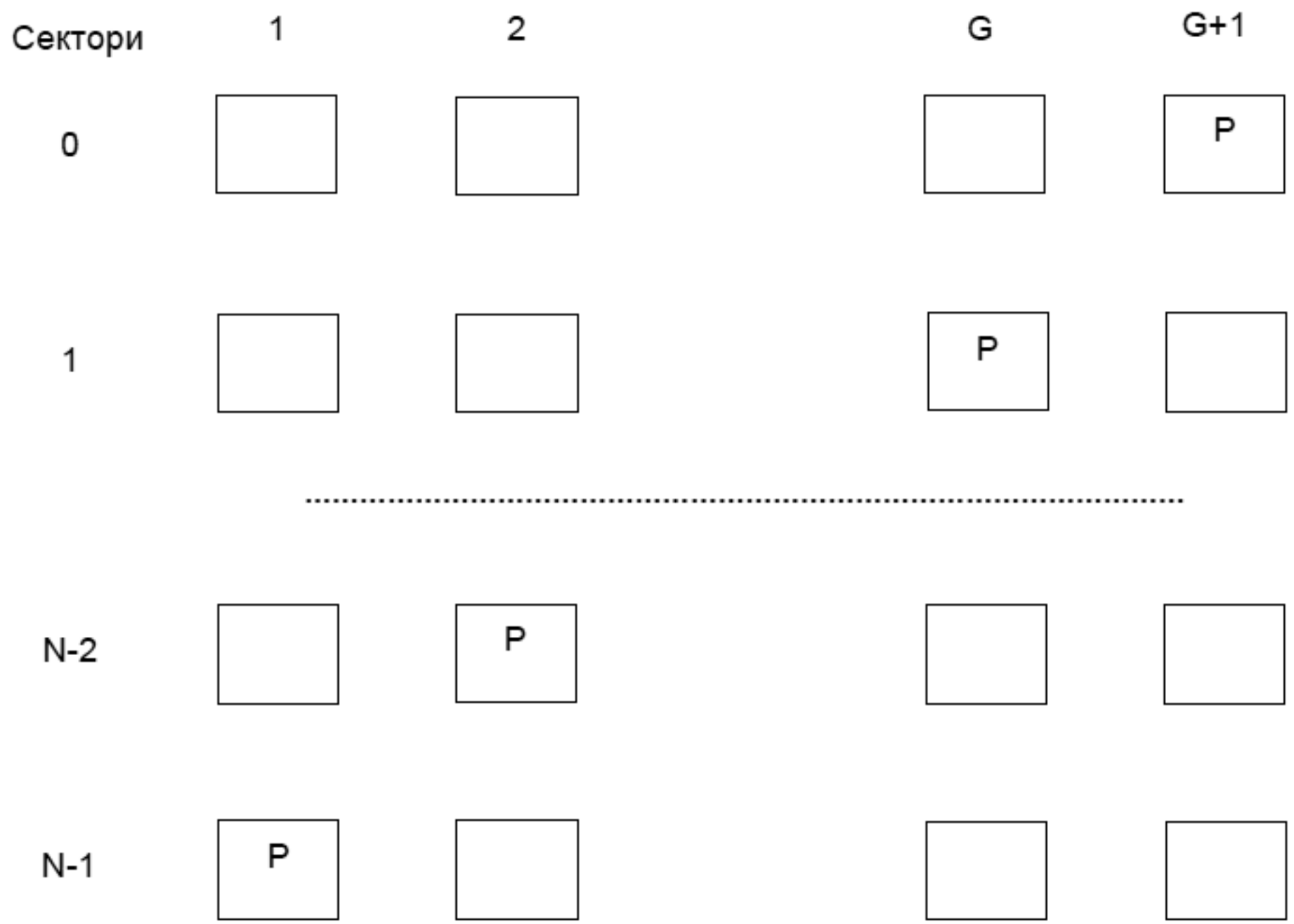
RAID 4 разделя данните на ниво блок върху G на брой дискове, като използва един диск за съхранение на контролната информация, т.е. C=1 както и при **RAID 3**. Контролната информация позволява да се възстанови пропадането на произволен единичен диск. Главното отличие между ниво 4 и ниво 3 се състои в това, че при последната разслоението се изпълнява на ниво сектори, а не на байтове. За определяне на новото значение на четността са необходими стария блок за данни, стария блок за четност и новия блок за данни, или:

нова четност=(стари данни **XOR** нови данни) **XOR** стара четност

Производителността на ниво 4 е много добра за четене (същата както ниво **RAID 0**). Обаче записите изискват контролната информация да се записва всеки път. Това забавя малките по обем записи, с произволен достъп, макар че за големите по обем записи или последователните записи, процесът е сравнително бърз. Понеже само един диск в матрицата съхранява данните с излишък, цената на тази конфигурация се оказва сравнително ниска.

RAID 5: Разслоение по блокове и по контролни суми / Няма контролен диск.

RAID 5 е една от най-популярните конфигурации. Тя е подобна на **RAID 4** но контролната информация е разпределена между всичките дискове. Така се премахва тясното място на **RAID 4** – единственият диск за контролна информация. Това е показано на фиг.10-1.

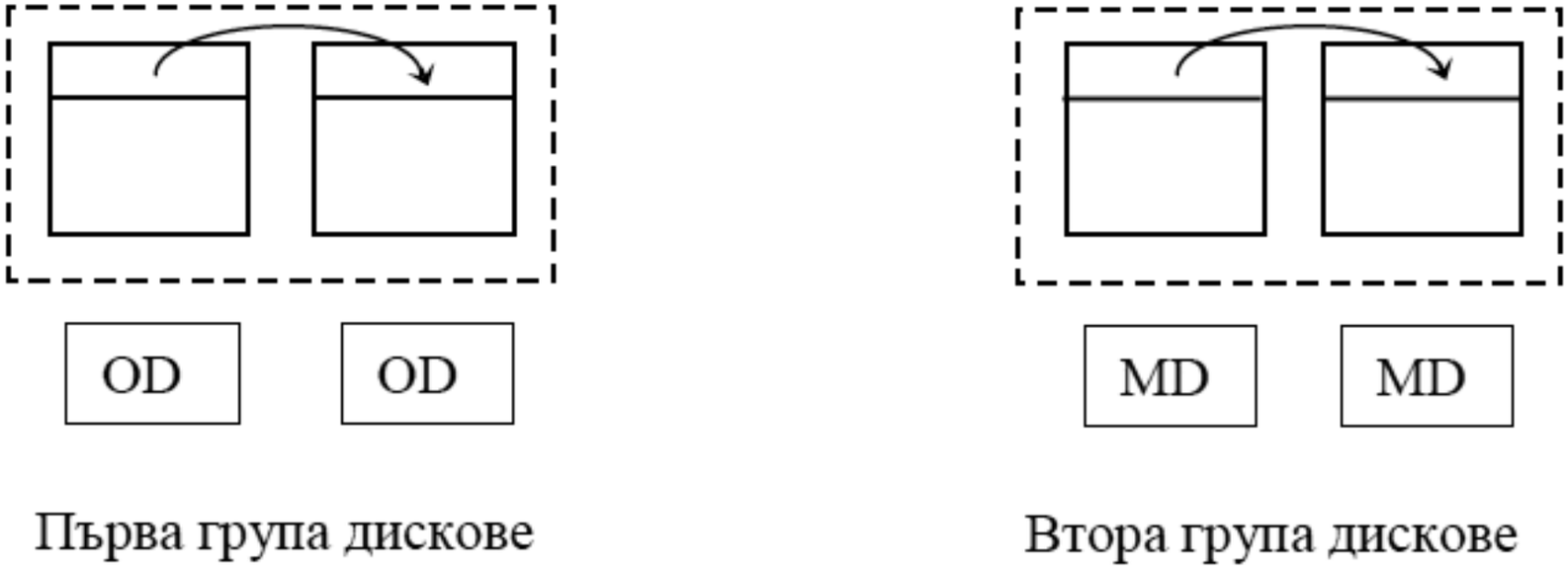


***RAID 6* Разслоение по блокове с двойно разпределение на контролните суми**

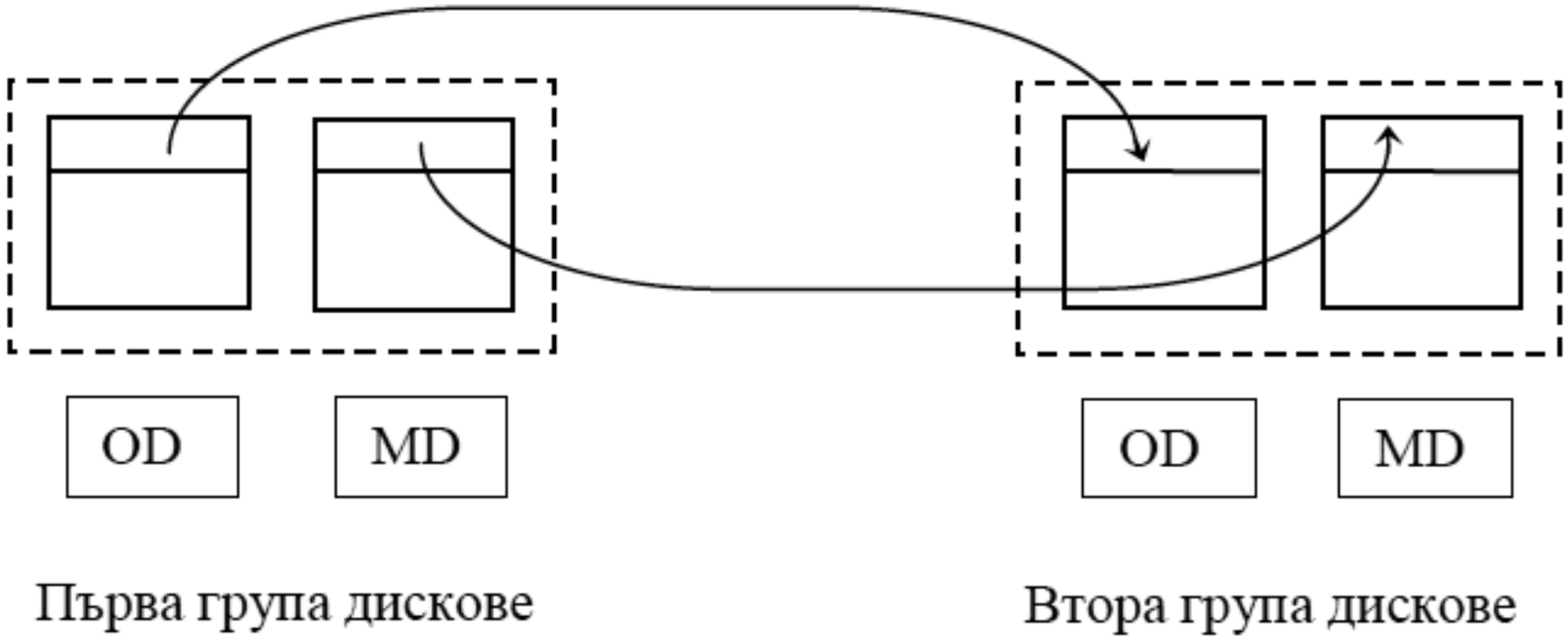
Разгледаните до сега нива коригират отказ само на един диск. ***RAID 6*** коригира отказ на произволни два диска в матрицата. Матрицата представлява обединение на дискове в двумерен масив, така че секторите се явяват трето измерение. Тук може да се осигури контрол по четност по редове, както е при **RAID 5**, а така също и по стълбове, които от своя страна могат да се разслояват за осигуряване на възможност за паралелен запис. При такава организация може да се преодолеят всякакви откази на два диска и много откази на три диска. Но при изпълнение на логически записи реално стават шест обръщения към диска: за старите данни, за четност по редове и по стълбове, а така също и за запис на новите данни и новите значения на четността. За някои приложения с изключително високи изисквания по отношение на устойчивостта на откази, такъв излишък може да се окаже приемлив, но при традиционните суперкомпютри и за обработка на транзакции този метод не е подходящ.

***RAID* 7 Асинхронно, кеширано разслоение, със специализирано управление**

За разлика от другите нива, **RAID 7** не е отворен индустриален стандарт. Това е търговски термин на **SCC** (**S**torage **C**omputer **C**orporation) за да опише своя собствен дизайн. **RAID 7** се базира на концепциите на **RAID 3** и **RAID 4**, но до голяма степен преодолява някои от ограниченията на тези нива. В частност включването на голямо количество кеш памет, аранжирана на няколко нива и специализиран процесор, работещ в реално време за управление на дисковата матрица. Тази хардуерна поддръжка позволява на матрицата да поддържа много независими операции, увеличавайки производителността от всички видове, и да поддържа висока устойчивост на откази. В частност, **RAID 7** предлага голямо усъвършенстване на производителността при случайни операции за четене/запис. Но получаваните резултати са за сметка на цената. Това е решение, което за сега се поддържа от една компания.



а) организация **RAID 0+1**



б) организация **RAID1+0**

3.3. Сравнение между различните нива.

RAID 0 е най-бързата и ефективна от всички матрици, но тя не осигурява надеждност

RAID 1 е подходяща при критични изисквания за производителност и същевременно осигурява устойчивост на откази. В допълнение, минималното количество дискове, които са необходими за изграждането й са само 2.

RAID 2 е рядко използвано днес конфигурация, защото всички модерни дискове имат вградена **ECC** схема.

RAID 3 не позволява множество входно/изходни операции да се препокриват и изисква синхронизирано въртене на дисковете за да се избегне деградация на производителността при къси записи.

RAID 4 няма предимства пред **RAID 5** и не поддържа множество независими операции за записи.

RAID 5 е най-добрият избор за многопотребителска среда, която не е чувствителна към производителността при запис. Но най-малко 3, типично 5 диска са необходими за изграждането на тази конфигурация.

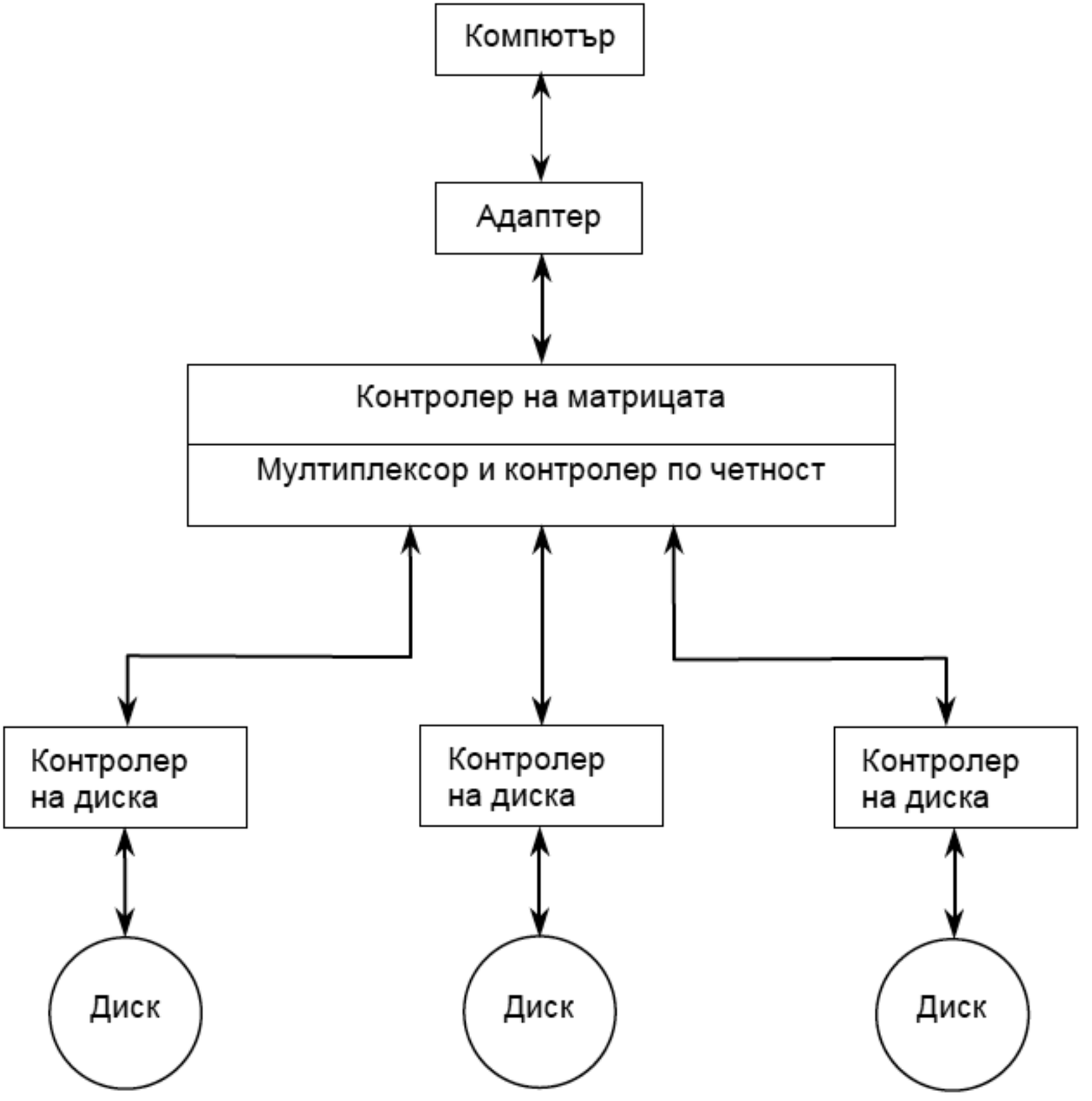
RAID 6 дава възможност за откриване и отстраняване на всякакви откази на два диска и много на три, но се явява една от най скъпите организации.

3.4. Програмна или апаратна реализация на RAID

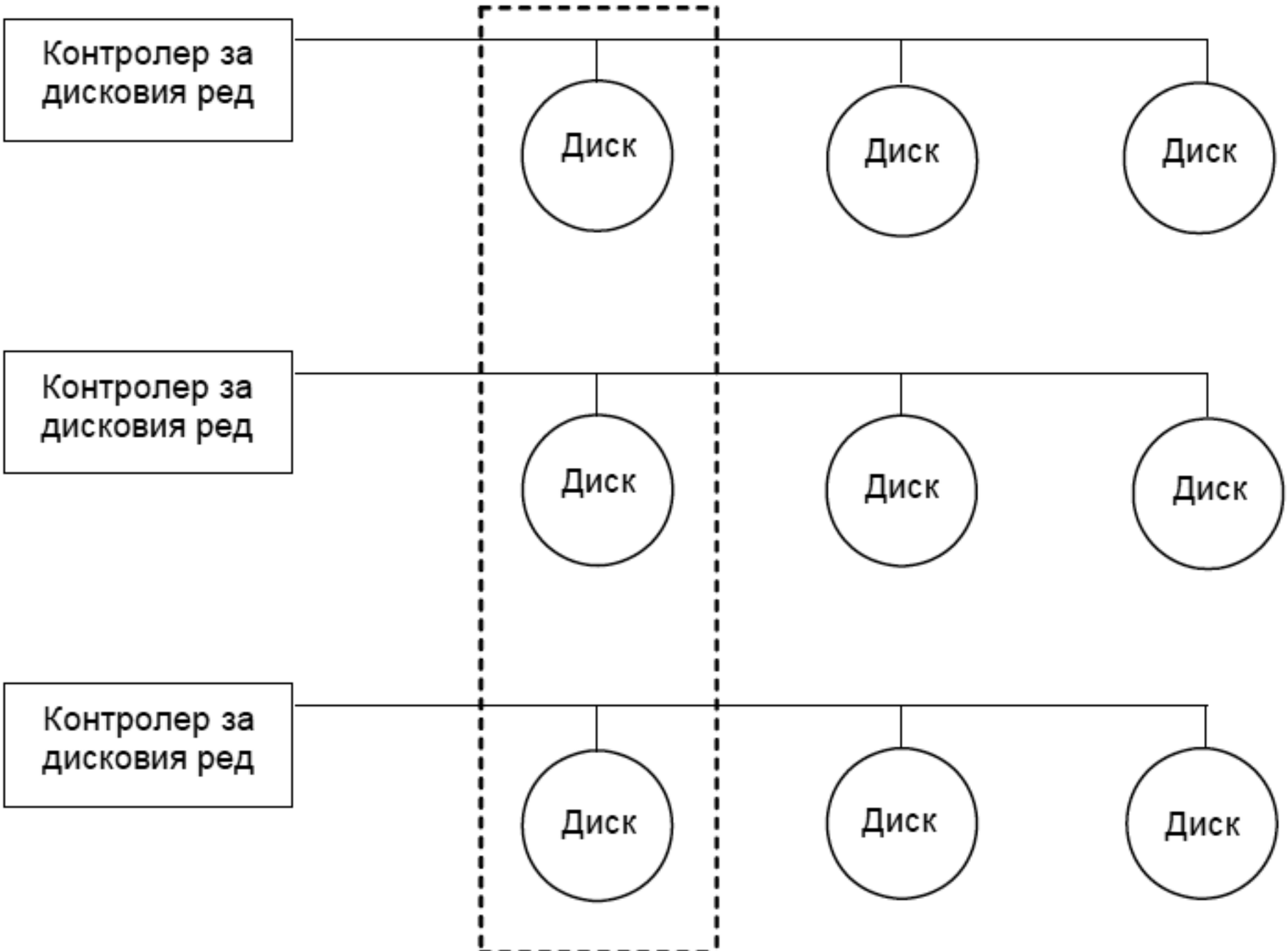
Програмната реализация е по-евтиното решение и е подходящо за малки системи. На системният администратор е предоставена почти цялата конфигурационна тежест за реализацията. В някои случаи не е възможно да се реализират изцяло предимствата, заложени в даденото ниво **RAID**. Например, реализацията на **RAID 1** върху един физически диск дава възможност да се възстановят загубените данни върху един сектор или пътечка, но не и ако пропадне управлението на диска като цяло.

Ясно е, че апаратно базираната **RAID** е по-скъпа от програмно базирана реализация. Но тя предлага следните предимства пред програмната реализация:

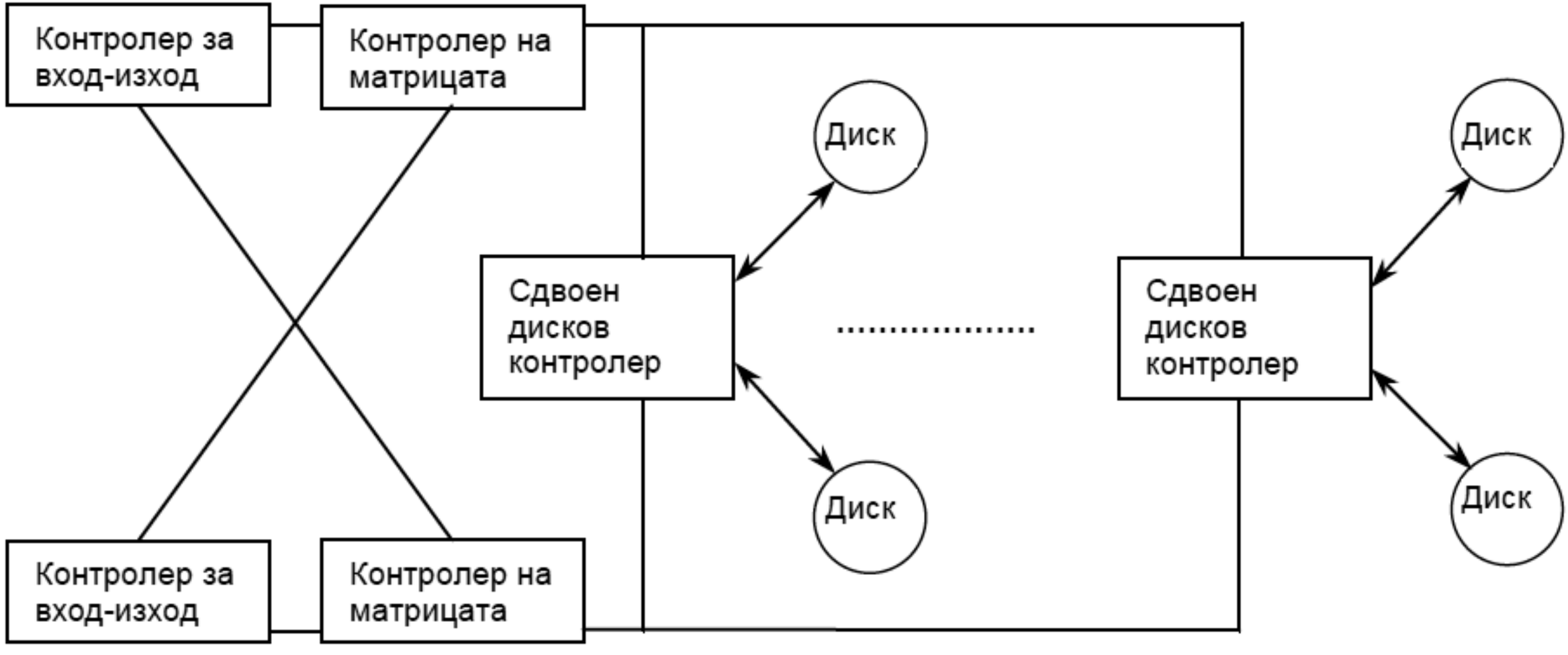
- лесно може да се извади и замени дефектиралия диск, без да се спира работата на системата;
- осигурява повече проверка за грешки и предлага конфигурационен контрол, за предпазване на администратора от грешки.



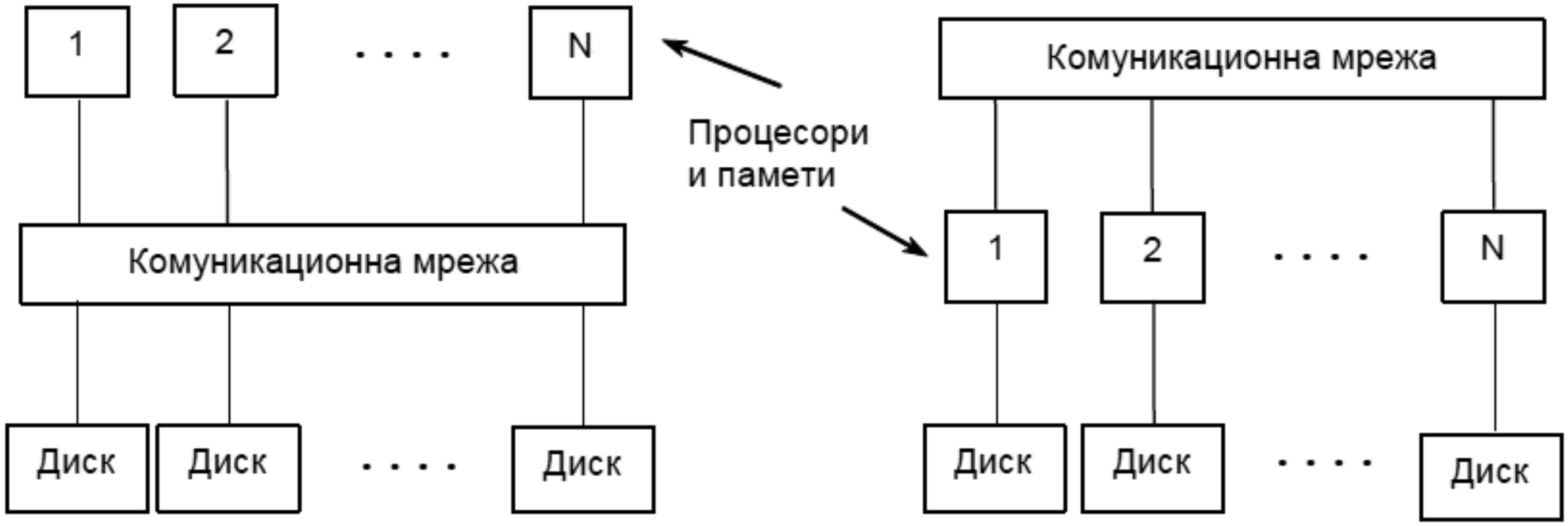
Фиг.10-3. Компоненти на контролера на дисковата матрица.



Фиг.10-4. Двумерна организация на матрицата



Фиг.10-5. Елементи на устойчивостта на откази матрица от дискове.



а) силно свързана система

б) слабо свързана система

Фиг. 10-6. Обобщени структури на дисковата подсистема на компютрите с разпределена памет.

Организацията на входно-изходната подсистема в паралелните компютри с разпределена памет поражда и някои специфични проблеми. Основните от тях са:

• Осигуряване на мащабируема производителност на входно/изходната подсистема. Това се налага от аналогичното изискване към паралелните компютри с разпределена памет. В случая производителността на всяко входно/изходно устройство се измерва чрез възможността му да поддържа работно натоварване на изчислителната област.

• Осигуряване и поддържане на балансирано натоварване. Въпросът за натоварването възниква от степента на съответствие между декомпозирането на данните на приложно ниво и разположението им по дисковата матрица, дефинирано от размера на лентата на декластеризация (striping size).

• Увеличен трафик на данните между изчислителните и входно/изходните възли. Този трафик зависи от приложната задача, метода за нейното решаване, топологията на комуникационната мрежа, осигуреното балансирано натоварване и др.

• Ширината на лентата на пропускане е ограничена от размера и броя на комуникационните канали между изчислителната област и входно/изходните устройства.

Край