Анотация

Учебникът е и е предназначен за студентите от специалност "Компютърни системи и технологии" към катедра "Компютърни науки и технологии" при ТУ - Варна. Това е второ допълнено и преработено издание и е съобразено с настоящата учебната програма. За усвояването на изложения материал са необходими предварителни познания върху основните устройства на компютъра и тяхното взаимодействие. Освен от студентите, учебникът може да се използва и от докторанти, както и от всички проявяващи интерес към паралелните компютри.

Учебникът фокусира вниманието върху различни архитектури на паралелни компютри. Обсъдени са техните особености при организацията на изчисленията, анализирани са предимствата и недостатъците на всяка една архитектура, както и по между им, а така и спрямо последователния компютър. Основно внимание е отделено на работата на процесора, но също така се обсъжда и организацията на паметта – вътрешна и външна. Там където е необходимо е обърнато внимание и на цялостното управление на компютъра.

Авторът изказва предварително благодарности на всеки за направените от него препоръки за подобряването му.

Авторът

СЪДЪРЖАНИЕ

СПИСЪК

На най-често срещаните съкращения
(по азбучен ред)

А. Български съкращения

ДКМ	Динамични Комуникационни Мрежи
КЕ	Комуникационен Елемент
КМ	Комуникационна Мрежа
ЛВ	Линия за Връзка
ОП	Оперативна Памет
ПЕ	Процесорен Елемент
ПЗ	Плаваща Запетая
СКМ	Статични Комуникационни Мрежи
ФЗ	Фиксирана Запетая
ФУ	Функционални Устройства
ЦП	Централен Процесор

Б. Английски съкращения

CISC	Complex Instruction Set Computer/Code	Процесор имащ сложен набор команди
MIMD	Multiple Instruction stream Multiple Date stream	Процесор с множество (потоци) от команди и множество (потоци) от данни
MISD	Multiple Instruction stream Single Date stream	Процесор с множество (потоци) от команди и един (поток) от данни
RAID	Redundant Array of Inexpensive Disks or Redundant Array of Independent Disks	Матрица от евтини дискове с излишък или Матрица от независими дискове с излишък
RISC	Reduced Instruction Set Computer/Code	Процесор, имащ съкратен набор команди
SCSI	Small Computer System Interface	Стандарт за прехвърляне на данни по шината между устройствата в компютъра
SIMD	Single Instruction stream Multiple Date stream	Процесор с един (поток) от команди и множество (потоци) от данни
SISD	Single Instruction stream Single Date stream	Процесор с един (поток) от команди и един (поток) от данни
TTA	Transport Triggered Architecture	Транспортно спускова архитектура
VLIW	Very Long Instruction Word	Свръхдълга команда
VLSI	Very Large Scale Integration	Интегрални схеми с голяма степен на интеграция

ТЕМА 1

ВЪВЕДЕНИЕ В ДИСЦИПЛИНАТА

Все още няма точно, ясно и еднозначно възприето определение на понятието *"архитектура на компютъра"*. Така например, до началото на 70-те, под понятието *"архитектура на компютъра"* се е разбирало описание на структурата на данните и регистрите, необходими за изясняване на набора команди на процесора и интерпретацията им. С други думи казано, това понятие обхваща онзи минимум от познания, които могат да представят програмата на асемблерен език. Но разпространението на виртуалната памет, а така също и на различните средства за разширение на вход/изхода се явява причина за увеличаване номенклатурата на блоковете на компютъра, познанията върху които, дават възможност за съставяне на ефективни програми. Така че познанията само върху командния набор не е достатъчно за описание на цялостното функциониране на компютъра. Едно съвременно схващане на архитектурата на компютъра е структурна организация на компютъра във вид на съвкупност от функционални модули и определяне на връзките между тях. Това определение доста добре се съчетава със съвременната тенденция за изграждане на компютъра чрез свръх големи интегрални схеми, като всяка една схема може да се разглежда като функционален модул.

Понятието *"архитектура на компютъра"* може да бъде определено и като разпределение на функциите, реализирани от компютъра на отделните нива и точно определяне на границите между тези нива. По такъв начин, ако за архитектурата на компютъра е определено някакво ниво, то първо е необходимо да се установи какви системни функции цялостно или частично изпълняват компонентите на системата, намиращи се над и под това ниво. След решението на тази задача трябва точно да се определят интерфейсите на разглежданите нива. От казаното до тук следва, че архитектурата на компютъра предполага организация на няколко нива. На фиг. 1-1 е показан компютър в неговия вертикален разрез. Тя се отнася за конвенционална система, т.е. за система с последователно действие.

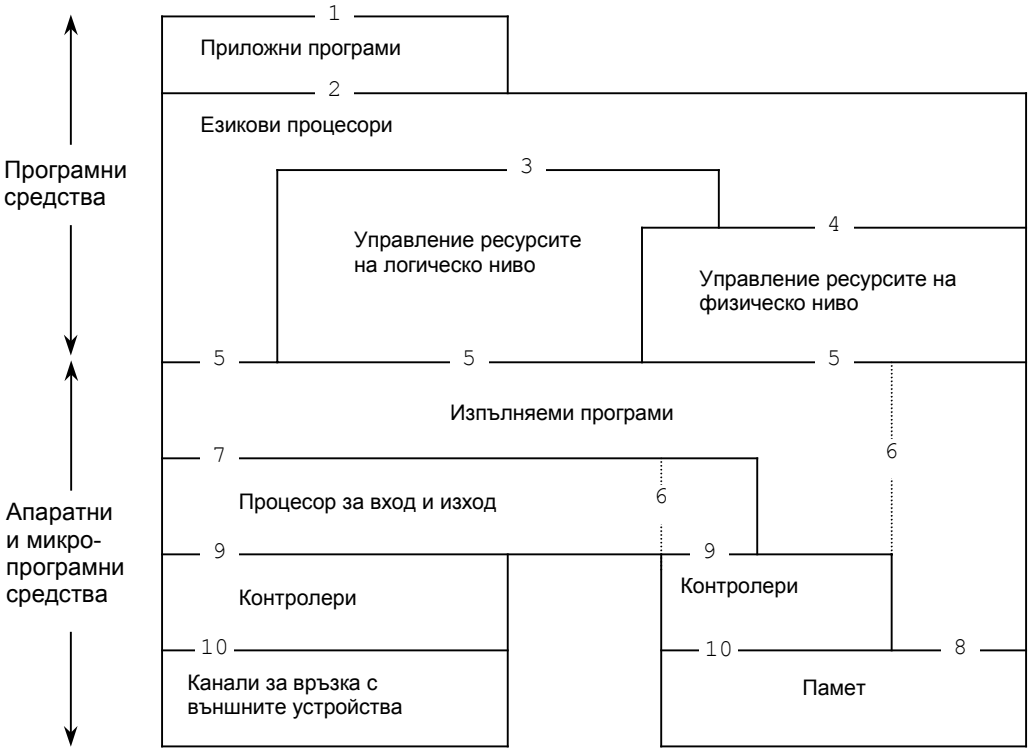
Нека накратко проследим тези нива.

Първо ниво определя какви функции по обработката на данните изпълнява компютърът като цяло и какви се възлагат на "външния свят" (програмиста, оператора и т.н.). Компютърът взаимодейства с външния свят чрез два набора интерфейси:

- езици (езици за програмиране, езика на операционната система, езика за манипулиране на базата данни и т.н.);
- системни програми (програми за редактиране, сортировка, обновяване на информацията и др.).

Нива 2,3 и 4 разграничават определени нива вътре в програмното осигуряване. Те нямат общоприети наименования. Ето защо те се наричат обобщено "архитектура на програмното осигуряване". Ниво 2 това са транслаторите; ниво 3 включва

реализацията на такива функции като управлението на базата данни, файловете, виртуалната памет, мрежовата телеобработка и т.н.; ниво 4 – управлението на външната и оперативната памет, вътрешните процеси протичащи в системата и т.н.



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

Ниво 5 отразява границата между системното програмно осигуряване (операционната система) и апаратното осигуряване. (Терминът "апаратно осигуряване" в случая обобщава както електронните схеми, така и микропрограмното осигуряване). Така това ниво позволява да се представи физическата структура на компютъра независимо от способа на реализация. Разграничаване на функциите, изпълнявани по-нагоре или по-надолу от това ниво е една от съставните части при разработката на компютърната архитектура.

Ниво 6 представлява интерфейса на микропрограмното управление, понеже функциите на всеки процесор и контролер на външно устройство могат да бъдат разпределени между микропрограмите и комбинационните схеми, и последователността от логически схеми. Ниво 6 отразява съгласуването на потока данни и управляващите сигнали с формата на микрокомандите. Много често ниво 6 и ниво 8 се наричат архитектура на процесора или организация на процесора.

Ниво 7 отразява какви функции реализира централния процесор, изпълняващите програми и какви процесорите за вход/изход. Ако процесорът за вход/изход отсъства в

конкретната конфигурация на компютъра, то неговите функции трябва да се разпределят между ниво 6 и нива 9 и 10.

Ниво 8 отразява интерфейса между централния процесор и паметта.

Нива 9 и 10 определят разграничаването на функциите реализирани от процесорите за вход/изход и контролерите. Ето защо те се наричат "архитектура на физическия вход/изход".

Приведеният по-горе общ обзор на архитектурата на различните нива на организация на компютъра, позволява да се определи архитектурата като абстрактно представяне на физическата система от гледна точка на програмиста, разработващ програми на асемблерен език или на проектанта на компилатора на език от високо ниво

От изложеното до сега следва, че ако ниво 5 се придвижи по-нагоре (според фиг.1-1), то делът на програмното осигуряване ще се намали за сметка на апаратното осигуряване, т.е. повече функции на компютъра ще се реализират по апаратен път. Обратно, при движение на ниво 5 надолу повече функции ще се реализират по програмен път. Така се получават два компютъра с едни и същи функционални възможности, но с различни характеристики.

Представеният на фиг.1-1 вертикален разрез на системата съответства на класическия, фон Нойманов компютър. Един хоризонтален разрез ще отразява архитектурата на паралелния компютър. В този случай трябва да се отразяват функциите между група от процесори, в някои случаи с различно предназначение, и съответстващите им интерфейси, чрез които те взаимодействат.

Архитектурата не се занимава с въпроси от управлението и предаването на данни вътре в процесора, конструктивните особености на логическите схеми или технологията на тяхното производство.

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.

Трябва да се подчертае, че процесът на проектирането на архитектурата на компютъра има итеративен характер и обикновено се налага многократно да се повтарят отделните или всички етапи на проектирането.

В последно време процесът на разработка на компютри съществено се е изменил. Проектантите отдавна са престанали да бъдат просто инженери по апаратни средства или програмно осигуряване. Сега ,те е необходимо не само да се ориентират в новите технологии, но и в приложната математика, мрежите от компютри, интерфейсите и системите за изчисления.

Преди всичко, компютрите от новото поколение трябва да бъдат гъвкави, както в смисъл на технологията, така и в смисъл на архитектурата. Те трябва да бъдат съвместими със новите технологии, а така също и да са разширяеми. Още една черта на новите разработки, от персоналните компютри до суперкомпютрите, се явява и способността да образуват мрежи. Времето на отделно стоящите компютри е минало и е необходимо да се отчита, че разработваният компютър ще се включва в стандартна мрежа. Стандартите в многото области на разработка на изчислителни системи заемат важно място, защото те позволяват да се свързват в едно цяло оборудването на различни фирми. Това има смисъл не само за мрежите и апаратната част, но и за програмното осигуряване.

Литература

1. Г. Майерс
Архитектура современных ЭВМ, в 2-х книгах, Москва, Мир, 1985., т.1 стр.10-17.
2. J.Till
Computer system architecture. “Computer Design”, 1989, 37, No1, pp.50-54, 56, 58, 60, 62-63.

ТЕМА 2

ОСОБЕНОСТИ В АРХИТЕКТУРАТА НА СЪВРЕМЕННИТЕ КОМПЮТРИ

Независимо от голямото разнообразие на компютърни архитектури, всички те се базират на принципи, предложени от Джон фон Нойман още в началото на 40 години на миналия век. Тези принципи са:

- принцип на програмното управление;
- принцип на съхраняването на програми в паметта.

Независимо от развитието на технологията на производство и на програмирането през всичките близо 60 години, тези принципи остават същите (разбира се допълнени), и днес продължават да бъдат ръководно начало. Естествено въплъщаването на тези принципи става по различен начин спрямо компютрите произвеждани да речем преди 30 или 40 год.

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.

В исторически аспект погледнато, първите процесори и базираните на тях компютри са били от типа **RISC.** Но много скоро архитектурата е мигрирала към **CISC,** тъй като за повишаване на бързодействието на процесорите са се добавяли нови команди (апаратно или микропрограмно), приближаващи се към примитивите на езиците от високо ниво.

Повторното появяване **RISC** процесорите става в края на 70-те и началото на 80-те год. на 20-тия век. Теорията на компютрите със съкратен набор команди е разработена от Д. Петерсон от университета Бъркли, Калифорния. Отправна точна за развитието на тази теория се явява факта, че в традиционната архитектура (**CISC** компютрите) голяма част от времето на процесора (от 60-80%) се използва за изпълнение на неголямо множество команди. Тези

команди са за запис на операндите от паметта в регистрите и обратно, условен и безусловен преход, извикване на подпрограми. 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).

Нека да разгледаме до какво водят посочените по-горе изисквания и как са реализирани те.

Изисквания а) и б) позволяват да се изключат ред сложни проблеми, стоящи пред проектантите на апаратната част и на компилатора. Освен това се опростява управлението на процесора. Като следствие на това площта, която заема управляващото устройство върху кристала намалява значително. За сравнение при процесор MC 68000 (това е първият 32 битов процесор, разположен в един чип и има архитектура **CISC**) управляващото устройство заема 50% от площта на чипа, докато при един от първите **RISC** процесори RISC I, този процент е 6%! Като допълнително предимство може да се посочи намаляването на времето за разработка на управляващото устройство -средно два пъти.

Простотата на командните формати осигурява в частност, просто и бързо декодиране. Възможно е да се използват по сходен начин едни и същи апаратни блокове за изпълнение на почти всички команди. Така, изпълнението им може да се осъществи не чрез микропрограми, а чрез логически схеми. Всичко това в крайна сметка съкращава времето за тяхното изпълнение.

Изисквания в) и г) определят наличието на голямо количество регистри и използването само на команди от типа "регистър-регистър" за изпълнението на всички операции, освен операциите четене/запис от/в паметта. Както е известно,операциите "регистър-

регистър” се изпълняват 2-4 пъти по-бързо от операциите с кеш паметта и 8-12 пъти по-бързо от операциите с оперативната памет. Така, че с изпълнение на изисквания в) и г) се постига също увеличаване на бързодействието на процесора. За ефективното им изпълнение е необходимо регистрите да бъдат достатъчно на брой. При първоначалните разработки на компютри с **RISC** архитектура, броят на регистрите е бил сравнително голям, например RISC II има 138 регистъра, AMD 29000 – 192, Pyramid - 528 и т.н. Големият брой регистри е бил продиктуван от първоначалната липса на компилатори, достатъчно добре използващи регистрите на процесора. Но големият брой регистри създава проблеми със тяхното управление, а така също и заемат по-голямо място върху кристала. По-късно броят на регистрите е намален. Причините за това са две:

а) Проведените допълнителни изследвания показват , че с 8 регистъра се получава 80% от бързодействието, в сравнение с използването на неограничен брой регистри.

б) Разработени са и разпространени регистрово оптимизиращи компилатори, които ефективно използват наличните регистри в процесора.

Разбира се **RISC** архитектурата има някои негативни страни, които трябва да се имат предвид. Основните от тях са следните:

Регистровата архитектура открива възможности за намаляване бързодействието на паметта за данни с цената на повишена пропускателна способност на паметта за команди. Това свойство се оказва желателно при използване на кеш-памет за команди, която е по-ефективна от кеш-паметта за данни. Кеш паметта за команди е по-проста за реализация (изисква само четене) и при равни обеми вероятността за попадения в нея е по-висока отколкото в кеш паметта за данни, защото локалността на обръщенията за команди е по-висока отколкото за данните. Така че няма **RISC** процесор, който да няма поне кеш-памет за команди.

Простотата на използване на команди в компютрите с **RISC** архитектура осигурява висока скорост на тяхното изпълнение. Повишаването на скоростта на изпълнение на отделните команди, като правило, превъзхожда повишаването на тяхната ефективност, защото за изпълнението даже на прости операции, реализирани с една единствена **CISC** команда, са необходими няколко **RISC** команди. Това означава увеличен размер на паметта за съхраняване на програмата, а от тука и увеличени обръщения към паметта, което в крайна сметка се отразява негативно на общата производителност.
И така, производителността на **RISC** процесорът е по-голяма при изпълнението на отделно взета команда в сравнение с изпълнението на цяла програма.

На края трябва да се отбележи, че прякото преобразуване на обектни текстове или асемблерски програми, разработени за традиционните компютри, в програми за **RISC** компютри е нецелесъобразно поради същественото различие на структурата на

командите. Пряката, по командна транслация води до значително увеличаване дължината на програмата, т.е. до заемания обем памет. Необходима е нова компилация на програмите с компилатор, предназначен за **RISC** компютър.

• 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.

В следващите теми на този учебник основно се дискутира прилагането на паралелната обработка в съвременните компютри.

• 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.

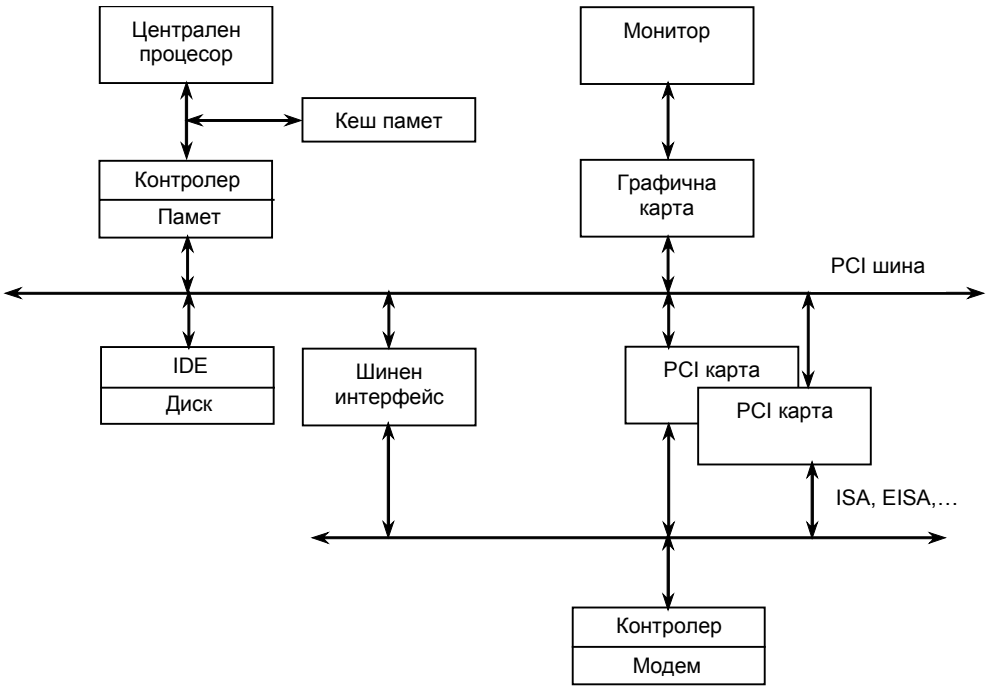
Кратка история на развитието на шината при персоналните компютри. През 1981 г. IBM въвежда първата системна шина за персоналните компютри, която е 8 битова за данни и работи на тактова честота 4,77 MHz. Тя свързва всички основни компоненти на компютъра: процесор, памет и входно-изходни устройства. Малко по-късно, 1984 г. тази шина е разширена до 16 бита и честота й е увеличена на 8 MHz. Понастоящем е известна под името **ISA** (**I**ndustry **S**tandard **A**rchitecture) и се явява най-разпространената шина през 80-те години на миналия век. Пропускателната й способност е 16 MB/sec.

През 1987 г. отново IBM въвежда нова шина **MCA** (**M**icro **C**hannel **A**rchitecture), която е несъвместима с периферните **ISA** модули. Тя е отначало 16 битова, а по-късно става 32 битова. През 1988 г. дебютира 32 битовата шина **EISA** (**E**xtended **I**ndustry **S**tandard **A**rchitecture), като промишлен вариант на **MCA** шината. Пропускателната и способност е 33 MB/sec.

През 1990 г. фирмата Compaq в своя компютър DeskPro 486/25 въвежда йерархия на шините – **Flex/MP** и по-късно **TriFlex**. При тях процесорната шина,към която са присъединени процесорът, кеш паметта и основната памет е отделена от **EISA** шината, към която се включени периферните устройства. Така се осигурява независима работа на отделните устройства на компютъра.

През 1992 г. асоциацията **V**ideo **E**lectronics **S**tandard **A**ssociation – **VESA** обявява техническа спецификация на локалната шина **VESA VL-Bus** като опит да се сложи край на хаоса съществуващ във видео локалните шини. Разширените слотове **VL-Bus** обикновено се използват съвместно със стандартните слотове **ISA** или **EISA.** Пропускателната способност на тази 32 битова шина е 133 MB/sec. Най-сериозният й недостатък е свързан със съвместяване на работата на принципа "включи и работи" (plug and play).

През 1993 се анонсира шината **PCI,** която е 32 битова и първоначално работи с тактова честота 33 MHz. В **PCI** шината е развита по-нататък идеята на Compaq за йерархия на шините в компютъра. Тя се разполага не върху самата процесорна шина – фиг. 2-1, а на едно междинно ниво между процесорната шина и периферните устройства.



Фиг.2-1. Йерархична организация на шината

Благодарение на това на екрана на монитора може да се изведе пълноцветно изображение с нормална скорост и висока разделителна способност с множество прозорци. Освен това, така се осигурява включването на допълнителни платки, които се конфигурират автоматично.

В съвременните персонални компютри, посочената на фиг.2-1 йерархична организация на шината намира практическа реализация чрез чипсета на компютъра и по-специално чрез северния мост към който се включват бързодействащите компоненти (процесор, памет и графична карта) и южния мост към който се включват останалите компоненти – външната памет, клавиатура, мишка и т.н.

Литература

1. Ст. Каблешков
Архитектура на ЦЕИМ. С., Техника, 1978
2. Т. Мото-ока
ЭВМ пятого поколения (Концепции, проблемы, переспективы). М., "Финансы и статистика", 1984.
3. Hindin H.J.
Fifth generation computing: dedicated software is the key, "Computer Design", 1984, 23, No10, pp.150-152, 154-160, 162, 164.
4. Цв. Петров
TriFlex, Computer, 7/93, 49-52.
5. Basert E.
RISC design streamlines high power CPUs. "Comput.Des.",1985, Vol. 24, No7, pp.119-122.
6. Hennessy J.L.
VLSI processor architecture. "IEEE Trans.on Copm.", 1984,33, No12, pp.1221-1246.
7. М. Амамия, Ю. Танака
Архитектура ЭВМ искусственный интеллект.М., Мир, 1993, стр.7-38.
8. У. Стиллингс
Архитектура компьютера с сокращенным наборам команд. ТИИЭР, т.76, No 1, январь 1988р стр. 42-63.
9. Всичко за Pentium. НИСОФТ ООД,1998, 83-123.
10. www.top500.org
11. Таслаков Цв.
Компютърни архитектури. Печатна база на ТУ Варна, 2001

ТЕМА 3

ВЪВЕДЕНИЕ В ПАРАЛЕЛНАТА ОБРАБОТКА

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 свидетелствуват 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 използуват 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 [1]. Ключът е в разпределената функция на генератора, който синхронизира протичащите операции. Днешните чипове използуват централизиран часовник и всички операции протичат на един и същи интервал или цикъл. В новия дизайн часовникът е децентрализиран и локално генерирани часовници отговарят за по-големи порции от потока на данни. Но независимо от всичко това, крайната скорост на разпространение на светлината, по-точно на електричеството, е основната причина за да се твърди, че този подход изчерпва своите възможности. В тази връзка трябва да се отбележи, че въведеният от Хокни критерий "полупроизводителност на компютъра" (това е дължината на вектора, за която производителността намалява наполовина) не се влияе от тактовата честота, а от архитектурата на компютъра [2]. Същевременно е налице значително усъвършенстване на традиционния компютър (най-значителното от което е въвеждането на конвейерната обработка), но така или иначе компютърът остава подчинен на фон Нойомановия модел на изчисление. Ето защо през последните години се отделя все по-голямо внимание на втория подход за увеличаване на производителността, а именно чрез внасянето на радикални промени в организацията на изчислителния процес, т.е. реализирането на паралелни компютри.

Първата идея за паралелни изчислени е по-стара с 100 г. от компютъра и е дадена от генерал Л. Менебрия, съратник на Ч. Бабидж и датира от 1842 г. Интензивни изследвания започват обаче в края на 50-те и началото на 60-те г. на миналото

столетие, но едва с появата на **VLSI** и по точно на микропроцесорите, започва практически да се реализират тези идеи.

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.

2. Нива на паралелност

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.

И в двата случай за реализацията е необходимо компютър с няколко процесора, всеки от който изпълнява свой набор

команди. Процесорите могат да взаимодействуват помежду си или чрез общо поле на паметта или чрез обмен съобщения.

- 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.

В реалният свят е възможно в един и същ компютър да се използва паралелност на повече от едно ниво. Най-често това се постига като се съчетае реализацията на паралелност в пространството (паралелна работа на няколко процесора) с паралелност във времето (конвейерната обработка на данни и команди). Така се мултиплицира ефектът от въвежданата паралелност.

3. Оценка на производителността (по основният индекс бързодействие)

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

$$S = \frac{T_1}{T_P} \qquad (3.1)$$

където: S е коефициент на изменение на бързодействието;

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

T_P е времето за решаване на същата задача на паралелен компютър с P на брой процесора, имащи същите характеристики както тези на процесорът от еднопроцесорния компютър

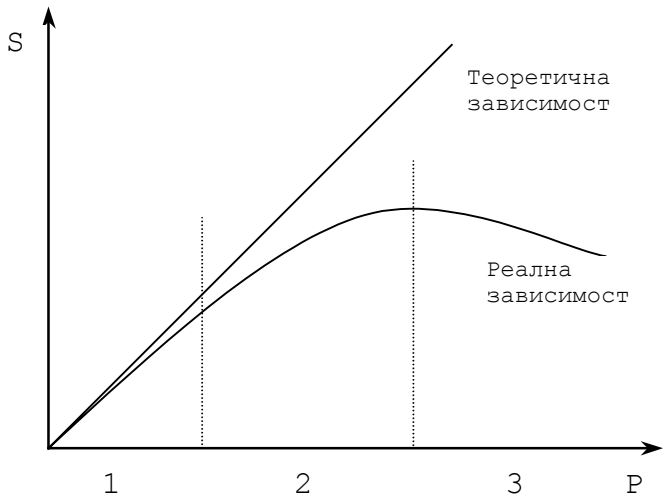
За да има смисъл от въвеждането на паралелна обработка, очевидно е, че трябва да е в сила отношението **S>1**. Формула (3.1) показва относителното изменение на производителността на компютъра, след въвеждане на паралелна обработка. Така, този коефициент отчита архитектурните особености и не зависи от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

Ефективността от въвеждането на паралелна обработка се дава чрез:

$$E = \frac{S}{P} \le 1 \qquad (3.2)$$

Коефициентът **E** показва средното натоварване на всички процесори, включени в паралелния компютър. Колкото този коефициент е по-близък до 1, толкова повече процесори работят през цялото време на решаване на задачата и обратно.

Обобщената зависимост на **S(P)** е дадена на фиг.3-1. За идеалният компютър с увеличаване на **P** следва пропорционално увеличаване на **S**. В реалния свят нещата се различават съществено. На фиг.3.1 условно могат да се разграничат три зони - 1, 2 и 3-та. В първата зона, увеличаването на **S** се доближава, повече или по-малко, до идеалното изменение на **S**.



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

Във втората зона, пропорционалното увеличение на **S** постепенно намалява, докато се достигне до насищане. В третата зона **S** не само че не расте с увеличаване на **P**, но и постепенно намалява. Разликите между идеалното и реалното **S** се дължат на специфични задачи, които трябва да се решат от паралелния компютър и по-точно разпределение на задачата между процесорите, синхронизация на множеството процесори, разрешаване на конфликтни ситуации при достъпа до общи ресурси и т.н. Ясно е, че решаването на тези задачи изискват допълнително време, което се сумира към времето за изчисление. Очевидно, най-доброто използуване на паралелните компютри е в първата зона и отчасти от втората. Когато това е постигнато се казва, че паралелния компютър е мащабируем т.е. производителността нараства пропорционално (или близко до пропорционалното) с увеличаване на броя процесори.

4. Модели на мащабируемостта. [6,8]

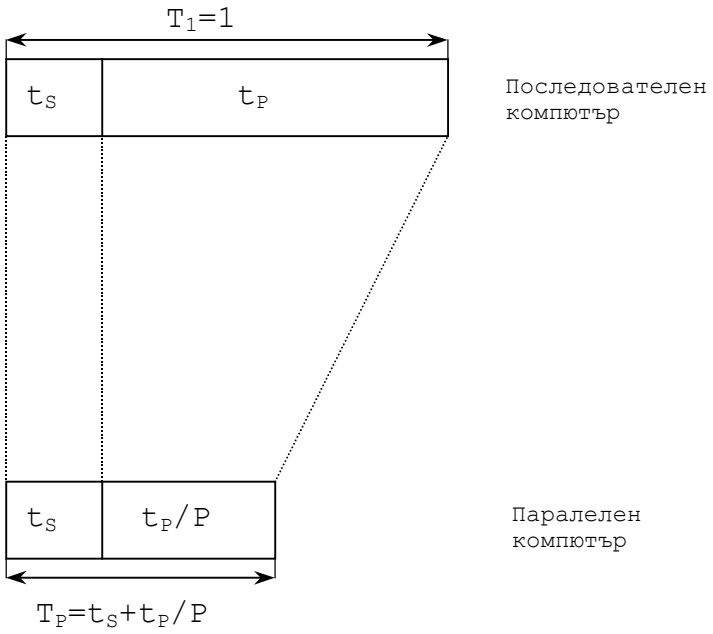
Тука ще дискутираме два от основните модели за мащабируемост на паралелните компютри.

4.1. Ограничение на задачата по размер

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

$$S = \frac{T_1}{T_P} = \frac{t_s + t_p}{t_s + \frac{t_p}{P}} = \frac{1}{t_s + \frac{t_p}{P}} .$$

Ясно е, че с увеличаване на **P** 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

Например, ако **P**=1024 и $\mathbf{t_s}$=0.01 (т.е. само 1% от програмата се изпълнява последователно), то **S**=91.18 и **E**=0.089 (това е доста песимистичен резултат).. Според този модел производителността расте основно до степен, която се определя от задачата, по-точно от нейните възможности за паралелно решение, а не от броя използвани процесори.

4.2. Ограничение на задачата по време.

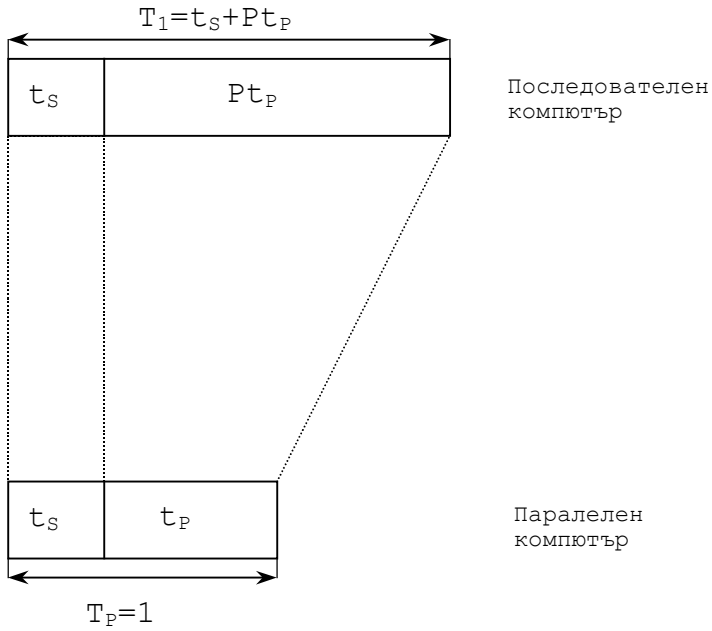
През 1987 г. Barsis предлага алтернатива на закона на Amdahl.

При закона на Amdahl се предполага, че $\mathbf{t_p}$ е независимо от **P**, 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 $\mathbf{t_s+Pt_p}$ - фиг.3-3 и за **S** се получава:

$$S = \frac{T_1}{T_P} = \frac{t_s + Pt_p}{t_s + t_p} = t_s + Pt_p .$$

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

5. Класификация на паралелните компютри

Всяка класификация трябва да притежава следните качества [10]:

а) Възможност да класифицира всичките съществуващи компютърни архитектури, а така също и да предвиди нови такива.

б) Диференциация на съществено различните изчислителни структури.

в) Еднозначна класификация на всеки компютър.

В голяма част от съществуващите схеми за класификация тези правила не са изпълнени. Някои твърде груби категории, използвани в такива схеми, не позволяват да се включат някои достатъчно жизнеспособни структури и еднозначно да се класифицират различните форми на конвейеризация.

• Класификация на Флин [2, 4, 10]. Една от първите и все още най-популярна класификация на компютрите е предложена от 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.

Преди да се посочат проблемите на **SIMD** компютрите, трябва да се подчертае, че понятието "процесор", което е еднозначно определено при компютрите с фон Нойманoвска архитектура, при паралелните компютри е размито. Различни автори [2-12], много често под понятието "процесор" включват различни по сложност компоненти, започвайки от прост изчислителен елемент, изпълняващ само основните аритметични операции, без възможност за извличане и декодиране на командите и достигат до напълно самостоятелен и независимо функциониращ компютър. В контекста на изложението на конкретна архитектура става ясно какви са функционалните изисквания към процесора.

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.

Решаването на една голяма част от тези проблеми се дискутират по-подробно в теми 7, 8, 12 и 13.

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.

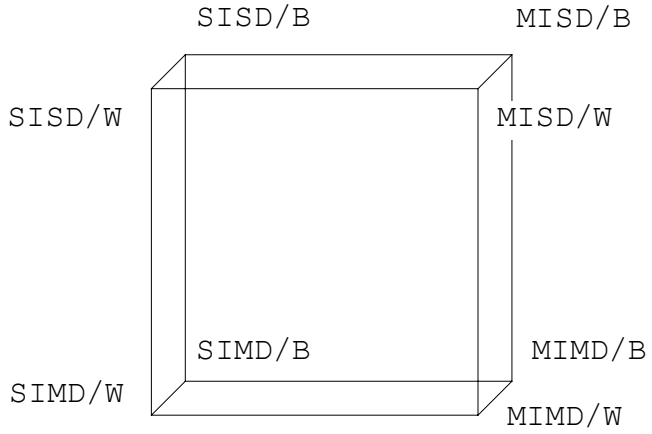
в) Осигуряване на методи за динамична реконфигурация на ресурсите на системата за задоволяване на постоянно променящите се изисквания и заетост на процесорите.

Отново решаването на част от проблемите е дискутирано по-подробно в теми 9, 10, 12 и 13.

Класификацията на Флин е елегантна по своята простота и симетрия и обхваща всички видове компютърни организации. Схемата несъмнено е издържала изпитанието на времето, най-вероятно поради своята ортогоналност. Същевременно недостатъка на тази класификация се състои именно в това, че тя е прекалено широка. В един и същ клас попадат компютри с твърде различна архитектура. Това е така защото класификацията се осъществява по достатъчно широка функция (по потока информация циркулиращ в компютъра), а не по архитектурни признаци. Независимо от всичко това, тази класификация все още е най-широко използвана.

За подобряване на класификацията са се предлагали различни допълнения и изменения на основната, Флинова класификация. Едно от тях използва два нови признака, отчитащи

начина на обработка – по битове или по думи. Така се получават 8 класа – четири класа за обработка по думи и четири класа за обработка по битове. Това схематично е показано на фиг.3-4.



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

Друг опит е направен от Foutain [11] и отчита такива особености на архитектурата на паралелните компютри като типа на автономията (операционна, адресна и мрежова), топологията на свързване на процесорите и форматът на данните – Таблица 3-1.

Таблица 3-1.

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.

Разбира се, съществуват и други способи за класификация на компютрите с паралелна архитектура несвързани с тази на Флин, а именно класификация по функционално предназначение, по способа на управление, по степен на свързаност на отделните процесори в компютъра, по степен на равноправие и т.н. По-долу съвсем на кратко ще се спрем на тези класификации.

• 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).

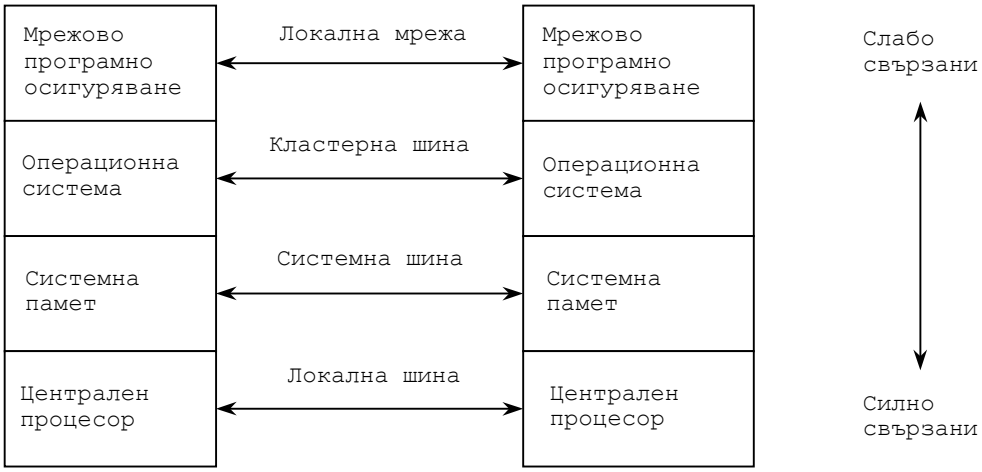
в) С управление по потока заявки (demand driven).

В компютрите от първия тип всяка команда определя следващата подлежаща на изпълнение команда. В компютърът с управление по потока данни, наличието на необходимите аргументи стартира изпълнението на командата, използваща тези аргументи. В компютрите с управление по заявки, командата се изпълнява, когато нейния резултат е нужен за изпълнение на друга команда. На тези въпроси е посветена тема 11.

От посочените три класа компютри най-голямо разпространение са получели тези от първият клас. В зависимост от степента на независимост на работата на отделните процесори, тези компютри се подразделят два типа архитектури – архитектура от тип SIMD и архитектура тип MIMD. Типични представители от първия тип се явяват различните матрични процесори, използвани за научни и инженерни изчисления. Компютрите с архитектура MIMD, в която всеки процесор в съответствие със своя програма обработва свой поток от данни, притежават значително по-голяма мощност и гъвкавост, отколкото компютрите с архитектура тип SIMD.

• Класификацията по степен на свързаност на процесорите е показана схематично на фиг. 3-5.

В силно свързаните системи, процесорите (два или повече) са обединени в обща шина. При наличието в системата на един или два специализирани процесори, интензивния обмен на данни по локалната шина между процесорите не води до забележимо снижаване на производителността на системата. Но при увеличаване броя на процесорите, локалната шина става тясно място в системата. Използването на двувходови памети, свързани с единия си вход към локалната шина, а с другия към системната шина позволява до известна степен да се снизи интензивността на обмена на данните по локалната шина. При добро балансиране на интензивността на обмен на данните чрез двата входа и използването на прости алгоритми за разпределение на натоварването между процесорите, такава архитектура се оказва достатъчно ефективна.



Фиг.3-5. Класификация по степен на свързаност

Ако интензивността на потока данни между оперативната памет и "своя" процесор съществено превишава интензивността на потока данни между тази памет и другите процесори в системата, по-ефективно се оказва използването на система, състояща се от множество автономни компютъра, всеки от които работи под управлението на собствена операционна система. А най-слабо свързана система се състои от множество компютри, обединени от локална мрежа, информацията по която с предава във форма на пакети или съобщения.

• 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.

В заключение трябва да се каже, че съществуват изключително голямо разнообразие от класификационни схеми и един и същ паралелен компютър може да се окаже в различни класове в зависимост от избрания параметър за класификация.

6. Разпределена обработка.

Отделно разглеждане заслужават и изчислителните мрежи, които се създават от териториално обособени центрове и могат да образуват разнообразни изчислителни схеми. Изчислителните центрове функционират автономно и взаимодействат помежду си чрез канали за връзка. Изчислителните мрежи се отличават

съществено от другите типове паралелни изчислителни системи и имат ярка изразена специфика както по отношение на организацията на функционирането, така и по отношение на програмното осигуряване.

Различието между разпределените изчислителни системи (дискутирани в тема 10) и изчислителната мрежа е въпрос, свързан с разположението и функционирането на възлите, а също така и на начина на взаимодействие между отделните апаратни и програмни ресурси. Ако всичките компоненти на системата са относително зависими и локализирани, т.е. те могат да се считат като части на едно цяло, системата е разпределена такава. Ако компонентите са достатъчно сложни и независими, те се разглеждат като различни машини (компютри). Посредством канали за връзка и съответно апаратно и програмно осигуряване може да се изгради мрежа от изчислителни машини. Подобна мрежа също притежава разпределен изчислителен ресурс, но той не е локално разположен.

Първите опити за така нареченото "разпределено изчисление" датират от 70-те години на миналия век, когато започват да се създават първите мрежи от компютри [12]. Идеята е проста – множество компютри (няколко милиона), включени към някаква мрежа (напр. Интернет) да бъдат натоварени с решаването на една задача представляваща глобален интерес за цялото човечество – напр. намирането на извън земен разум. Това се осъществява чрез специално създадена програма, натоварваща процесорът, когато той е свободен. Тя е с максимално нисък приоритет и се активира единствено когато процесорът бездейства. Тя е невидима за потребителя и фактически не се отразява върху работата на компютъра. Освен това, в общия случай не оказва почти никакво влияние върху мрежовия трафик.

Информацията, която трябва да бъде обработена, се намира на сървъри в Интернет [13]. След като програмата установи връзка с Интернет, тя поема част от задачата по обработката. Общата задача се разпределя между отделните компютри, които са се включели в проекта – всеки от тях извършва самостоятелни изчисления, различни от изчисленията на другите компютри, и изпраща обратно резултатите си и взема нови задачи при всяко включване в Интернет. При това не е необходимо компютрите да бъдат през цялото време свързани в мрежата.

Кога е удачно да се използва "Интернет компютър" и кога суперкомпютър? Отговорът е прост: когато задачата е ограничена по размери е по-добре да се използва разпределена обработка, а когато е ограничена по време – централизирана.

Повече информация може да се намери на следните адреси:
http://www.distributed.net
http://www.mersenne.org
http://setiathone.ssl.berkeley.edu

Литература

1. Personal Computer World. март, 2000, стр.34.
2. Р. Хокни, К.Джесссхоуп.
Параллельные ЭВМ. М., “Радио и связь”, 1986.
3. Moknoff N.
Parallelism makes strong bit for next generation computers. “Computer Design”, 1984, V.23, No 10, pp. 110, 112-114, 116-118, 120-124, 126-131.
4. Artym R., Mason J.S.
XPXM/C: taxonomy of processor coupling techniques. “IEE Proc.”, 1988 E 135, №3, pp.173-180.
5. Schindler Max.
Multiprocessing system embrace both new and conventional architectures. “Electron. Des”, 1984, Vol.32, №6, pp. 97-106, 108-110, 112, 114, 116, 118,120,122,124,126,128.
6. Gustafson J.L.
Reevaluating Amdahl’s Law. “ Communications of the ACM”, 1988, Vil.31, №5, pp.532-533.
7. Gallant J.
Parallel processing users in a revolution in computing. “EDN”, 1988, Vol. 33, №18, pp.86, 89-94, 96, 98, 100.
8. Jaswinder Pal Singh, John L. Hennessy, Anoop Gupta
Scaling Parallel Programs for Multiprocessors: Methodology and Examples. “Computer”, 1993, July, pp.42-50.
9. Gordon Bell
Ultracomputers: A teraflop Before Its Time. “Communications of the ACM”, 1992, Vol.35, №8, 27-47.
10. Д.Ивенс
Системы параллельной обработки. М., “Мир”, 1985, стр.8-46.
11. T.J.Foutain
A Review of Parallel Computer Architectures. International Seminar on Parallel Computers, Programming, Reliability and Diagnostics, 18-21 May 1989, Varna, Bulgaria.
12. К.Боянов, В, Кисимов, А. Петков, А. Терзиев,П. Русев
Разпределено управление на слабо свързани системи. С., Техника, 1989.
13. Коняров П., Александров А.
Интернет компютърът на бъдещето. PC WORLD No10, 2000, стр.14-16.
14. Jonah McLeon
Look out Cray: here comes a super priced supercomputer. EUSA, 1989, No.8,pp.83-86.
15. K.J.Thurber
Parallel Processor Architectures Part 1: General Purpose Systems. “Computer Design”, January 1979, pp. 89-97.
16. K.J.Thurber
Parallel Processor Architectures Part 2: Special Purpose Systems. “Computer Design”, February 1979, pp. 103-114.

17. Lowrence Curran
Chose the right parallel architecture, ED, 1989, No. 11,pp. 41-47.
18. Vasilii Zakharov
Parallelism and Array Processing. IEEE Transactions on Computers. Vol. c-33, No.1, January 1984, pp. 45-78.

ТЕМА 4

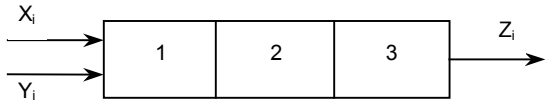
ОСНОВНИ ПРИНЦИПИ НА КОНВЕЙЕРНАТА ОБРАБОТКА

1. Въведение

Конвейерната обработка е едно от средствата за повишаване на производителността на процесора, чрез въвеждане на паралелизъм по време [1,2,3]. Въвеждането й се счита като естествено развитие на фон Нойманoвата архитектура. Конвейерната обработка се базира на разделяне на подлежащата за изпълнение базова функция на малки части, наречени подфункции и изпълнение на тези подфункции от отделни апаратни блокове, наречени степени. Подобно на движението на изделие по физически конвейер, командите и данните се преместват по степените на цифровия конвейер със скорост, която не зависи от неговата дължина (броя степени), а само от скоростта, с която новите обекти могат да се подават на входа на конвейера. Тази скорост от своя страна се определя основно от времето, за което един елемент може да бъде обработен от една степен на конвейера (обикновено най-бавната).

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.

На Фиг.4-2 е показана времедиаграмата на заетостта на степените в конвейера. Тази диаграма е построена при предположението, че всички степени на конвейера изразходват едно и също време за изпълнение на операциите на назначените им подфункции. Както ще видим по-нататък, това изискване не е

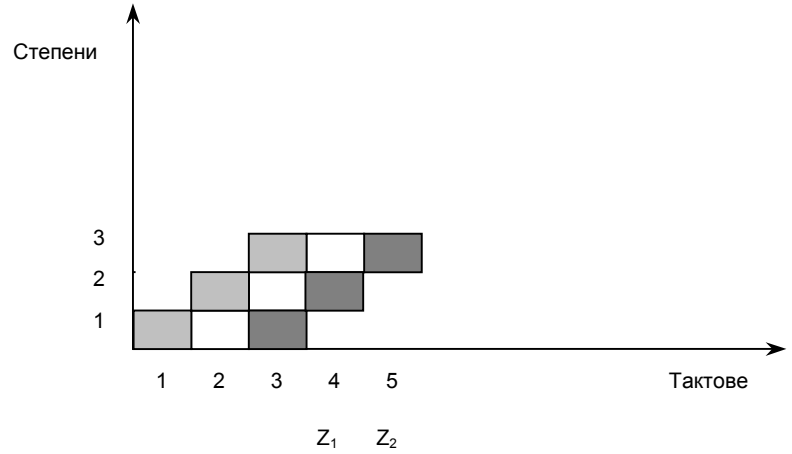
---

* Операцията събиране с плаваща запетая в различните процесори е разбита на различен брой под функции, напр. в IBM 360/91 на две под функции, в TI ASC - на шест под функции и т.н.

толкова тежко за изпълнение и не нарушава направените разсъждения.

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).



Фиг.4-2. Времедиаграма на заетостта на степените на конвейера.

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:

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

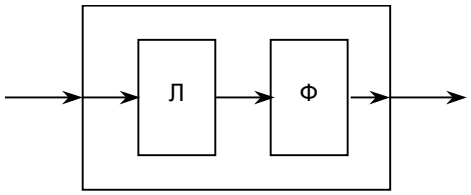
Ясно е, че с увеличаване на **n**, **S** се увеличава и се стреми към своята пределна стойност **L**.

От този прост пример се вижда, че конвейерната обработка е особено подходяща при обработката на дълги вектори.

След направените разсъждения до тук, може да се даде следното по-строго определение за конвейерна обработка: 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.



Фиг.4-3. Обобщена структура на произволна степен на конвейера.

Блок "логика" осъществява същинските логически операции, съответстващи на всяка подфункция. Блок "фиксатор" представлява свръхбърза памет и служи за подаване на данните в следващата степен в строго определен момент.

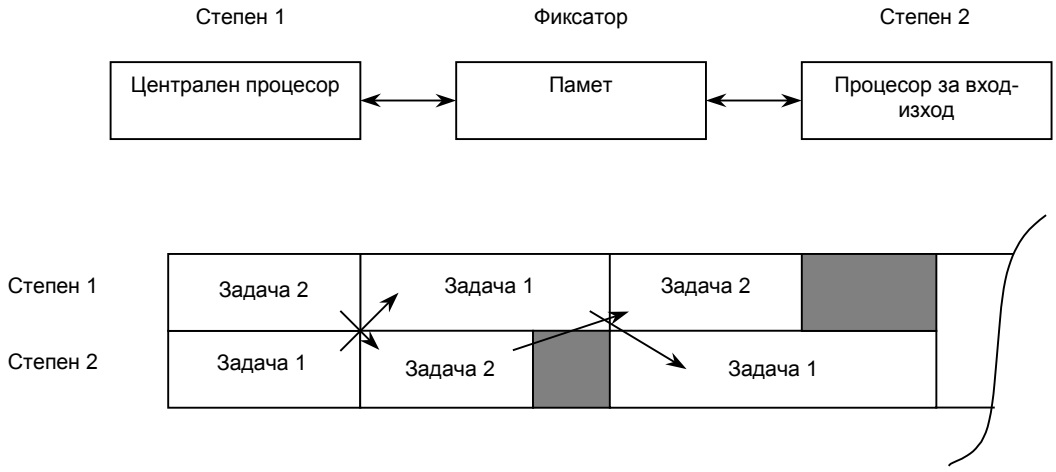
От изложеното по-горе става ясно, че конвейерът е синхронно изчислително устройство.

Ако поне едно от посочените условия в определението за конвейер не е изпълнено, не следва да се говори за конвейерна

обработка. По-правилно е да се говори за препокриване на операциите. И в двата случая крайният ефект е един и същ - намалява се времето за обработка, но този ефект се постига с различни средства. Термитът "препокриване" следва да се използува когато има място дори и едно от следните положения:

- 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.



Фиг.4-4. Пример за препокриване на операциите в компютъра. (Тъмните правоъгълници показват неизползваното време.)

На Фиг.4-4 централният процесор (степен 1) осъществява изчислителните операции, процесорът за вход-изход (степен 2) - всички входно-изходни операции, а оперативната памет може да се разглежда като фиксатор.

2. Видове конвейери

Конвейерите биват: еднофункционален и многофункционален. Еднофункционалният конвейер, както подсказва името му, е способен да изпълнява един тип базова функция. Многофункционалният конвейер е способен да изчислява функции от различни типове. В него, в допълнение към входа за данни, има управляващ вход, регулиращ действието на конвейера.

От своя страна, многофункционалният конвейер може да се класифицира по честотата, с която се изменя изпълняваната функция на конвейер. Така конвейерът бива със статична конфигурация

(статичен конвейер) и с динамична конфигурация (динамичен конвейер).

• Статичен конвейер. - Изменението на типа на изпълняваните функции са относително редки и те стават непосредствено под управлението на програмата на програмиста. Пример за такъв конвейер е аритметичния конвейер във векторния процесор.

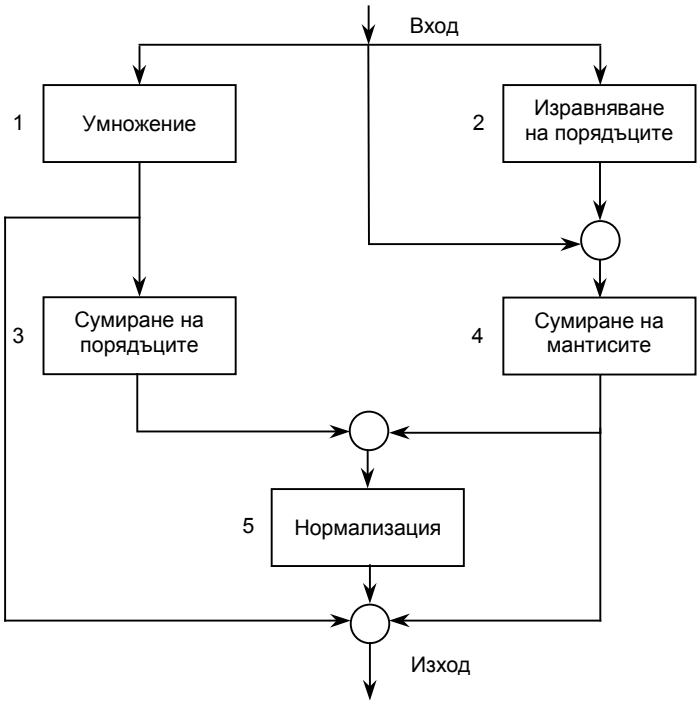
• Динамичен конвейер. - Изменението на типа на изпълняваните функции са чести, непрекъснати (динамични). Този тип конвейер се използува при изпълнение на командите на компютъра. Знаем, че по правило, всяка команда се отличава от предходните по формат и тип, и следователно е необходимо непрекъснато пренастройване на конвейера по отношение на изпълняваните функции. Понастоящем, в по-голямата част от процесорите се използува динамичен конвейер. Тука, в отличие от векторните процесори, фактическата реализация на конвейера много рядко се "вижда" от програмиста, а още по-малко се намира под негово управление.

Освен по честотата, с която се изменя изпълняваната функция, многофункционалните конвейери могат да се класифицират по пътищата по които данните се придвижват вътре в него. Така те могат да бъдат еднолинейни (еднопътни) и многолинейни (много пътни). При еднолинейните, потока от данни при произволна инициализация върви от една степен към следващата степен, независимо от типа операция. При конвейерите с множество от пътища, както подсказва наименованието им, има различни пътища при различните операции. Примерна структура на такъв конвейер е дадена на Фиг.4-5.

Този конвейер изпълнява следните операции: събиране и умножение на цели числа и събиране и умножение на реални числа. Със знакът **x** в Таблица 4-1 е показано използуването на всеки блок включен в конвейера за реализация на посочената операция.

Таблица 4-1

Блок	Сумиране на цели числа	Умножение на цели числа	Сумиране на реални числа	Умножение на реални числа
1		x		x
2			x	
3				x
4	x		x	
5			x	x



Фиг.4-5. Примерна структура на многофункционален, многопътен конвейер. (Кръгчетата на тази схема означават селектори, чрез които се избират пътищата в конвейера)

Основен недостатък на многофункционалните, многопътни конвейери се явява лошото използване на наличните апаратни блокове, включени в структурата на конвейера. Това ясно се вижда от Таблица 4-1 – няма операция, в която да се използват всичките блокове едновременно. Следователно, различните блокове са натоварени различно – напр. блокове 1, 4 и 5 са заангажирани едва 50% от времето, а блокове 2 и 3 – само 25%. Това, при равна вероятност за изпълнение на четирите аритметични операции, дава средно натоварване на блоковете на конвейера 40%.

3. Проблеми на конвейерната обработка

Увеличената скорост на изчисление води до проблеми непознати при "класическата" обработка. По-важните от тях са:

Прекъсванията. Системата от прекъсвания, която е достатъчно добре развита и ясна при класическата архитектура на фон Нойман, при високопроизводителните компютри и в частност при конвейерните търпи сериозно изменение и развитие. Основната трудност идва от това, че няколко фрагмента от операнди или команди едновременно се намират на различни стадии на изпълнение. Както е известно, прекъсванията са асинхронни спрямо нормалното изпълнение на програмата. Така трудността се състои първо да се определи къде трябва да се постави форсиран преход за обработка на прекъсването и второ как точно да се продължи прекъсната програма след

обработка на прекъсването. Трябва да се подчертае, че с нарастването на броя степени в конвейера (което означава като правило и по-висока производителност), се усложняват и задълбочават посочените по-горе трудности.

Задръжки в конвейера. Задръжките в непрекъснатата работа на конвейера се определят от структурата или използуването на конвейера и пречат за неговата работа с максимална скорост. Те биват: структурни и зависещи от данните.

Структурните задръжки възникват когато два различни фрагмента от данни се опитват да използуват една и съща степен в конвейера едновременно. По очевидни причини такива случаи се наричат стълкновение.

Като пример за възникване на такива задръжки, нека да разгледаме работата на многофункционалния конвейер, чиято структура е дадена на Фиг.4-5. Поради наличието на няколко пътя в конвейера, възниква специфичен проблем, а именно структурна задръжка или стълкновение. Например, нека конвейерът последователно да изпълнява операциите "сумиране с плаваща запетая" и "сумиране с фиксирана запетая". В първия такт - Фиг.4-5 операцията "сумиране с плаваща запетая" ще заеме степен 2. Във втория такт тази операция ще постъпи в степен 4. Но в същия този такт, по логиката на работа на конвейера, операцията "сумиране с фиксирана запетая" трябва също да заеме четвъртата степен. Настъпва конфликтна ситуация, която управляващият механизъм на конвейера трябва да предотврати.

За щастие, този тип проблеми подлежат на аналитично решение и могат да се отстранят още по време на проектирането на конвейера.

Задръжките зависещи от данните възникват, когато това, което протича в една степен от конвейера определя, могат ли данните да преминават през другите степени. Например, две различни степени трябва да използуват общата памет. Когато едната степен използува паметта, то другата, за която също е необходима памет, трябва да "работи" напразно, докато първата не завърши работата си. За разлика от структурните, тези задръжки изцяло зависят от данните и не се поддават на аналитично изследване.

Между командни зависимости. Този тип проблеми възникват при конвейерното изпълнение на команди. Едновременното изпълнение на съседни команди може да влияе на използуваните от тях операнди и изчислените от тях резултати. Типичен случай на такова влияние е когато i-та команда трябва да прочете някакъв елемент от данни, а i+1-та команда да го измени. С увеличаване на броя степени в конвейера, повече команди се намират в него на различен стадий на обработка и проблемът се задълбочава. Поради изключителната важност на проблема, методите за неговото решаване се дискутират по-подробно в Тема 5.

Проблем със запълването на конвейера. Както стана ясно, за да работи максимално ефективно конвейера, е необходимо всички степени в него да са натоварени. Ако дължината на обработвания вектор е съизмерима с броя степени на конвейера, или в програмата често се срещат команди за преход, то през по-голямата част от времето степените в конвейера няма да са натоварени или ще изпълняват ненужни за конкретния процес команди и от тук следва да се очаква намаляване на производителността на конвейера.

Литература

1. П.М.Коуги.
Архитектура конвейерных ЭВМ. М., “Радио и связь”, 1985, стр.9-94.
2. М. Амамия, Ю. Танака
Архитектура ЭВМ искусственный интеллект.М., Мир, 1993, стр.57-63.
3. Стоян Марков.
Изчислителни системи с висока производителност, С., Техника, 1990, стр.39-64.

ТЕМА 5

КОНВЕЙЕРНО ИЗПЪЛНЕНИЕ НА КОМАНДИТЕ В ПРОЦЕСОРА

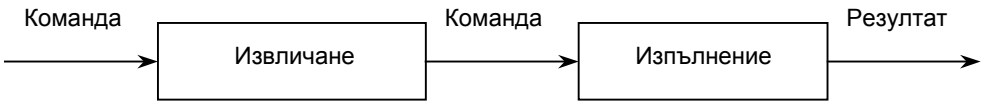
За програмиста, пишещ приложни програми, има две причини, поради които не следва да се отразява структурата на конвейера в системата команди. Тези причини са:

- Обикновено се определя единен набор команди за цялото семейство компютри, всеки от които ще изпълнява правилно едни и същи програми, независимо, че способите за реализация на командите могат да бъдат различни.
- По принцип е сложно да се проследява (и управлява програмно) всяка команда в конвейера.

Ето защо конвейерното изпълнение на командите в процесора остава скрито за програмиста.

1. Въведение

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.5-1.



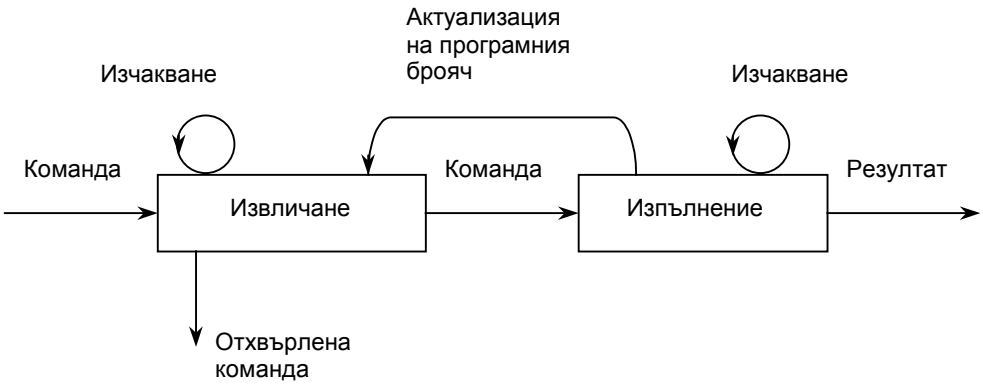
Фиг.5-1. Възможно най-простата структура на конвейер за команди

Така конвейерът има две степени: първата изпълнява подфункцията извличане, втората - изпълнение. Първата степен осъществява избор на командата и нейната буферизация. Когато втората степен се окаже свободна, първата степен прехвърля буферираната команда към втората степен. По време на изпълнението на командата от втората степен, първата степен използува незаетите цикли за обръщение към паметта за избор и буферизация на следващата команда. Описаната процедура е известна под името предварително извличане на 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.5-2.



Фиг.5-2. По-реалистичен модел на двустепенен конвейер

Загубата от време, предизвикана от втората причина може да бъде намалена чрез прогнозиране на следваща команда за изпълнение. Правилото е следното: избира се команда непосредствено следваща след командата за преход. Ако разклонението не се е състояло, загуба от време не съществува. Ако разклонението се е състояло, командата се отхвърля, а от паметта следва да се избере нова команда.

Независимо от всичко, така разгледания конвейер няма голямо ускорение. За увеличаването му се използуват 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.

2. Работа на конвейера

Работата на конвейера за изпълнение на командите ще бъде демонстрирана на базата на 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).

Изпълнението на тази програма ще проследим на **RISC** процесор, защото от една страна е по-лесно да се изложи конвейерното изпълнение на командите, а от друга защото и в реалния свят **RISC** командите се конвейеризират значително по лесно от **CISC** командите.

В този случай ще считаме, че всички аритметични и логически команди заемат две фази (напомняме, че операндите и резултата се съхраняват в регистрите):

IF - избор на командата;

E - изпълнение на командата.

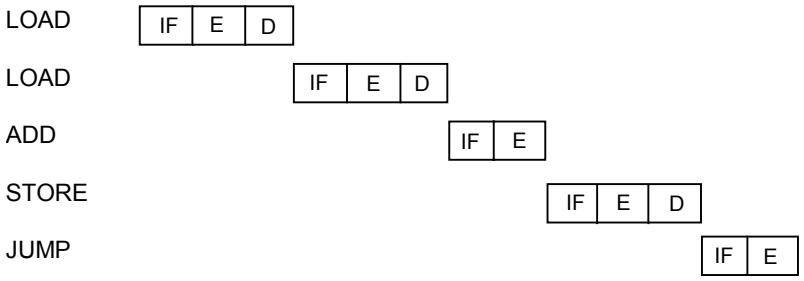
За изпълнението на операциите за четене/запис в/от паметта (**LOAD** и **STORE**) са необходими три фази:

IF - избор на командата;

E - изчисление на адреса на паметта;

D - обръщение към паметта.

На фиг.5-3 е показано "стандартното" т.е. не конвейерно изпълнение на програмния фрагмент.



Фиг.5-3. Не конвейерно изпълнение на примерна програма.

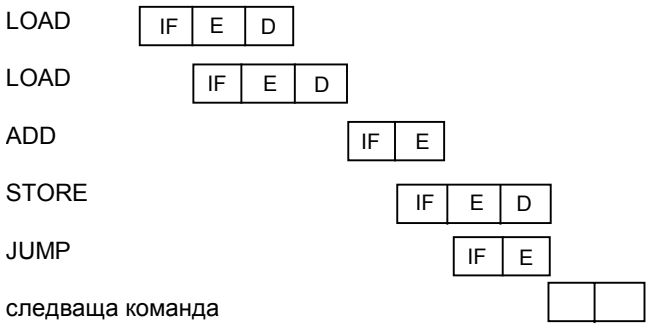
Лесно може да се установи, че изчислителният процес отнема 13 такта.

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 данните и командите.
- Б) Процесорът има отделни памети за данни и 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.



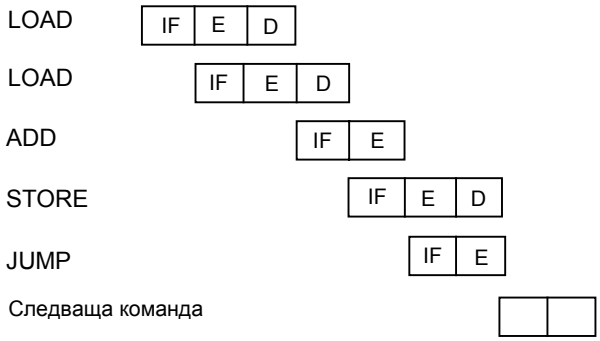
Фиг.5-4.Конвейерно изпълнение на програмата при наличието на обща памет за данни и команди

Както се вижда от фиг. 5-4 при изпълнението на някои команди (в случая **ADD**) не е възможно да се стартират в следващия такт, поради възникването на конфликтни ситуации. Затова командата **ADD** се задържа на два такта. Аналогична ситуация съществува и на края на програмния фрагмент – следващата команда трябва да се задържи на един такт за да може командата **JUMP** да промени съдържанието на програмния брояч.

Така разгледаното разписание изисква 8 такта, което означава едно повишаване на производителността с 62,5%.

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.5-5 е показано изпълнението на програмния фрагмент.

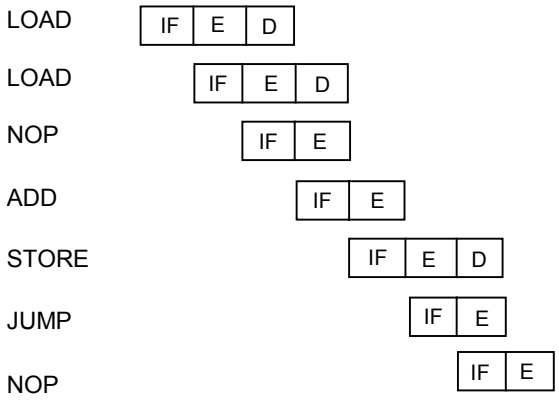


Фиг.5-5. Конвейерно изпълнение на програмата при наличието на отделни памети за данни и команди

И в този случай, ако се стартира в следващия такт командата **ADD** възниква конфликт. Тука конфликтът е по данни, защото съдържанието на регистър **B** все още не е установено с правилната стойност (фази **D** и **E** на командите **LOAD B,M2** и **ADD A,B** съвпадат!). Ситуацията с последната команда е аналогична с разгледания по-горе случай.

Така разгледаното разписание изисква 7 такта, което означава едно повишаване на производителността с 85,7%. Но трябва да се подчертае, че при това изпълнение е възможно съвместяване по време до 3 команди, което дава теоретично повишаване на бързодействието 3 пъти. Следователно този подход има много по-високи потенциални възможности от предходния (макар и да не се показват чрез този пример).

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.5-6.



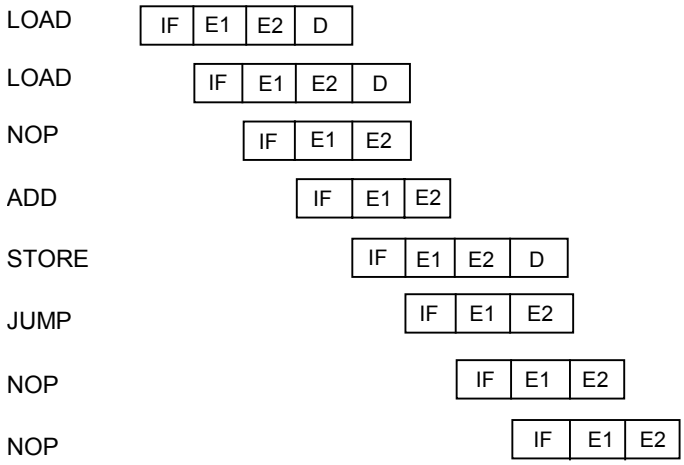
Фиг.5-6. Конвейерно изпълнение на програмата при наличието на отделни памети за данни и команди с използване на команда от типа NOP.

Тука вместо задръжка на командата **ADD** се включва нова команда **NOP,** която се явява трета поред. Нейната цел е да се даде възможност съдържанието на регистър **B** да се зареди с правилната стойност. Освен това в края на програмния сегмент е въведена още една команда – **NOP,** чието наличие се определя от следните съображения: Командата за преход **JUMP** нарушава последователния поток на изпълнение на командите, като променя съдържанието на програмния брояч. За да се даде възможност за правилно му установяване, при минимум апаратни средства, трябва да се включи нова команда - **NOP,** шеста поред в програмния фрагмент.

Независимо, че в случая има две нови команди спрямо оригиналната програма, времето за изпълнение е отново 7 такта.

Производителността на конвейера може да се увеличи допълнително, ако фаза **E** се раздели на няколко подфази: напр. две - фаза **E1** (напр. четене на регистровия файл) и фаза **E2** (напр. изпълнение на операциите в АЛУ). Това е резонно, защото фаза **E** има най-голяма продължителност от всички останали фази и така се създават предпоставки за намаляване на продължителността на такта на конвейера. Така се достига до следното разписание - фиг.5-7.

Тук формално се получава увеличен брой тактове – 10, но не трябва да се забравя, че тяхната продължителност е по-малка, в сравнение с тактовете в предходните примери. Ако се счита, че продължителността на такта е намаляла два пъти, то производителността се увеличава с 260%, т.е. 2,6 пъти в сравнение с не конвейерното изпълнение.



Фиг.5-7. Допълнително увеличаване на производителността чрез въвеждане на конвейер в изпълнителната степен

На базата на разгледания пример могат да се направят следните изводи: Конвейерното изпълнение на една програма е съпроводено с промени в нея, изразяващи се в появата на задръжки при изпълнение на някои от командите (концепцията на Университета Бъркли) или до вмъкване на нови команди от типа NOP (концепцията на Университета Станфорд). И в двата случая целта е да се избегнат възникналите междукомандни зависимости, така характерни при работата на конвейера.

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.

Предварителен избор на адреса на разклонение. В простия конвейер трудностите, предизвикани от командите за преход, се обуславят от това, че той трябва да избере една от двете команди, намиращи се на два различни адреса и той може да направи неверен избор. При тази техника, при постъпване на команди за условно разклонение, освен следващата след нея команда се избира и адреса на разклонението. По време на изпълнението на командата за условно разклонение става

запомнянето на този адрес. Ако трябва да се направи разклонението, то адресът на разклонението вече е избран и така се намалява времето за престой на конвейера.

Подходът с няколко потока команди за първи път е използван в компютрите на IBM System 370/168 и може да се разгледа като усъвършенстване на предходния. Очевиден изход е да се даде възможност на конвейера да избере двете команди, използвайки за това няколко потока. При този подход процесорът има два буфера за команди – основен и спомагателен. Основният буфер се използва за изпълнение на командите когато условието не е изпълнено, а спомагателния буфер се използва за паралелно изпълнение на последователност от команди, когато условието е изпълнено. При успешно взет преход, конвейерът само се превключва да изтегля команди от спомагателния буфер. Така той продължава да работи непрекъснато.

Прогнозирано разклонение се реализира в хода на изпълнение на програмата, като се запомнят в асоциативна памет най-често използваните преходи, срещани в командите за условен преход. По такъв начин, ако се изпълнява команда за условен преход, то се прави бърз достъп до асоциативната памет за да се извлече адреса на разклонението. Така се прави предварителен избор на командата следваща разклонението. Разбира се, при това се предполага, че вероятността за успешно вземане на преход се запазва същата, както и до този момент на изпълнение на програмата.

3. Междукомандни зависимости

3.1. Същност на междукомандните зависимости

В не конвейерните процесори всички операции, свързани с изпълнението на отделните команди, завършват до стартирането на следващата команда. По такъв начин фактическата последователност на изпълнение съответства на заложената от програмиста. При конвейерното им изпълнение това не е вярно, защото командите се препокриват по време. Така някои операции нужни за i+1, i+2...команди могат да се стартират и да се изпълнят до завършването на i-та команда. Това различие може да създаде трудности, ако не му е било отделено достатъчно внимание на етапа на проектиране, защото операциите, стартирани от i+1, i+2...команди могат да зависят от резултата на i-та команда. Съществуването на тази зависимост създава смущения, които намаляват максималната производителност на конвейера и следва да се избягват, което се прави при проектирането на конвейера или се анализират и отстраняват по време на изпълнение.

Съществуват 3 класа смущения между произволна двойка команди намиращи се в конвейера:

- **RAW** (**R**ead **A**fter **W**rite) - четене след запис;
- **WAR** (**W**rite **A**fter **R**ead) - запис след четене;
- **WAW** (**W**rite **A**fter **W**rite) - запис след запис.

По симетрия съществува четвърта зависимост - **RAR** (**R**ead **A**fter **R**ead), но тя не представлява конфликтна ситуация.

Едно по-строго определение за различните смущения може да бъде дадено на базата на 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:

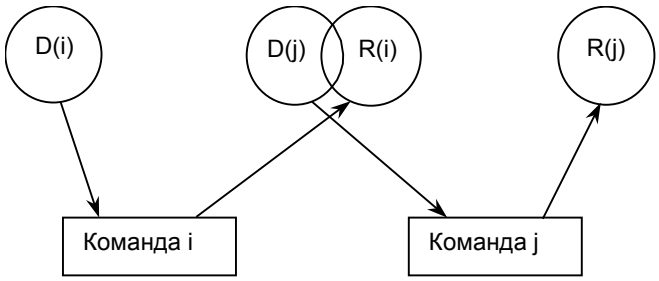
$R(i) \cap D(j) \neq O$ смущение **RAW**,
$D(i) \cap R(j) \neq O$ смущение **WAR**,
$R(i) \cap R(j) \neq O$ смущение **WAW**.

(считаме, командата j следва логически след командата i).

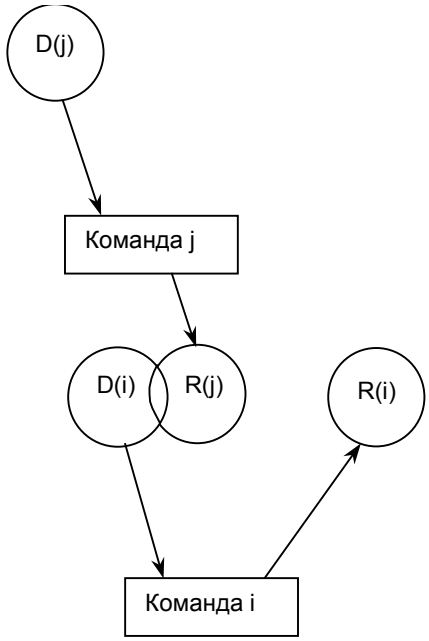
Смущение от типа **RAW** възниква когато j команда чете някакъв обект модифициран от i команда. Ако операцията в команда i, която модифицира този обект, не е завършила докато j се обръща към него, то тя ще прочете грешно съдържание. На фиг.5-8 е показана тази ситуация.

Смущение от типа **WAR** възниква когато **j** команда иска да модифицира някакъв обект, който се чете от команда i. На фиг.5-9 е показана тази ситуация.

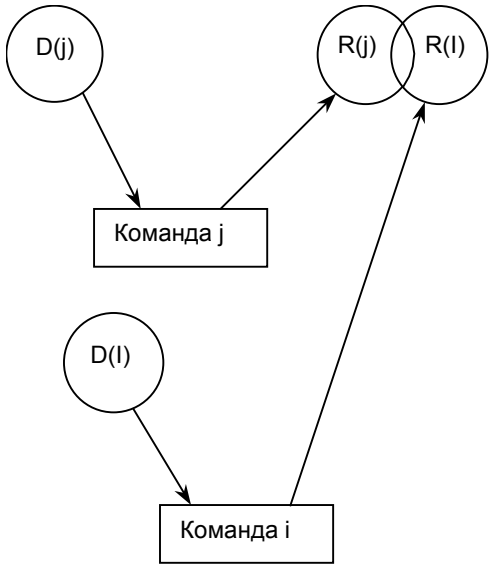
Смущение от типа **WAW** възниква когато двете команди i и j се опитват да обновят един и същ обект, но i-то запомняне може да настъпи след j-то. На фиг.5-10 е показана тази ситуация.



Фиг.5-8. Смущение от тип **RAW**



Фиг.5-9. Смущение от тип **WAR**



Фиг.5-10. Смущение от тип **WAW**

3.2. Откриване и отстраняване на междукомандните зависимости

Има два кардинални подхода за откриване и отстраняване на междукомандните зависимости - на апаратно и на програмно ниво.

3.2.1. 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 областта на определяне, или от областта на значения. 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.

3.2.2. На програмно ниво

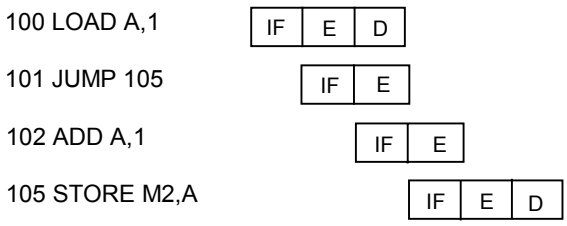
За отстраняване на споменатите зависимости между командите, се въвеждат команди от типа **NOP**, както беше посочено в т.2. Този способ се нарича отложено разклонение и спомага за увеличаване на ефективността на конвейера. Един друг способ е чрез реорганизация на програмния код и се нарича оптимизирано 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	

Причините за въвеждане на командата **NOP** при отложеното разклонение бяха изяснени в т.2. Тук трябва да се подчертае още веднъж, че в крайна сметка се увеличава обема на паметта, заеман от програмния сегмент.

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.

Разместването на командите е ефективно средство при оптимизация на работата на конвейера при безусловни разклонения и връщане от подпрограми. При условните разклонения се прилага друг подход: ако условието проверявано от командата за преход, може да бъде изменено непосредствено от предходната команда, в програмния код се вмъква команда **NOP.**

4. Кратки сведения за организацията на конвейерите в процесорите на Intel.

В тази точка ще се разгледат накратко особеностите на работа на конвейерите в процесорните архитектури P5 и P6 на Intel, както и на процесорът P4P. За по-подробно изучаване на работата на процесорите като цяло, следва да се обърнете към съответната литература [4,5,7,9].

Процесорната архитектура P5 е по-известна с името Pentium, а архитектурата P6 с имената Pentium II и Pentium III. Всяка една от тези архитектури има голям брой представители (Pentium, Pentium MMX, Pentium Pro, Celeron, Celeron A (Mendosino), Katmai, Coppermine, Xeon и т.н.), които естествено се различават помежду си, но тяхното ядро и по-специално работата на конвейерите се запазва.

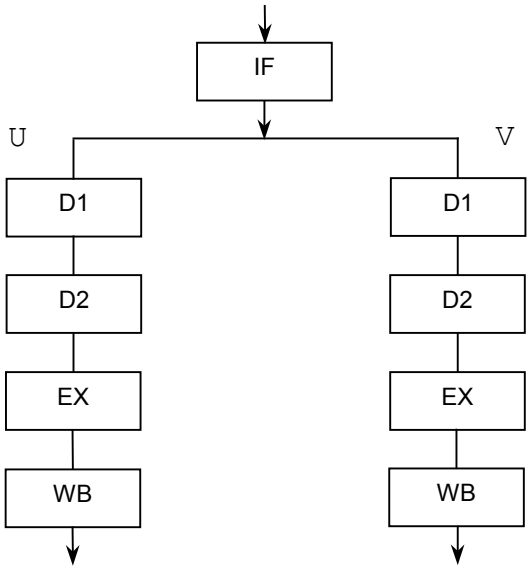
4.1. Архитектурни особености на конвейерите в P5.

Процесорната архитектура интегрира в един чип две устройства за целочислена аритметика, едно за аритметика с плаваща запетая, отделни 8K байтови кеш памети за код и данни, устройство за предсказване на преходите, както и устройство за управление на паметта. Процесорът е суперскаларен, т.е. при определени условия могат да се получават повече от един

резултат (в случая максимум 3). (Прието е да се казва, че процесорът има суперскаларна архитектура, когато в структурата му има повече от едно аритметични устройства работещи паралелно.)

4.1.1. Целочислени конвейери.

Процесорът има две устройства за целочислени команди, които са означени с **U** и **V**, и работят паралелно, т.е. той може да изпълнява две целочислени операции за един такт – фиг.5-12.



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).

Конвейерът **U** може да изпълнява всяка команда от фамилията x86, докато конвейерът **V** обработва само прости команди за цели числа, т.е. команди, които се изпълняват апаратно, и за които не е необходим микрокод, и командата **FXCHG** на устройството за числа с плаваща запетая. Когато командите се разпределят по двойки към конвейерите, командата за **V** винаги е след тази за **U**. Основните команди (MOV, и операциите с ALU) са реализирани апаратно, поради което се изпълняват значително по-бързо.

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 четвъртата 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.

Осигурено е, конвейерите **U** и **V** да започват и да завършат едновременно втора и трета фаза. Ако в тези фази изпълнението на някоя команда се забави, автоматично се задържа и изпълнението на командата в паралелния конвейер. Това означава, че командите в конвейерите **U** и **V** винаги навлизат едновременно във фазата **EX**. След като навлезе в нея, командата в конвейера **U** може да избърза, а командата в конвейера **V** – да се забави. В обратния случай командата в конвейера **U** трябва да спре, ако преди това командата в конвейера **V** с спряла. Не може друга команда да влезе в изпълнителната фаза на който и да е от конвейерите, докато предишните не са навлезли в последната фаза **WB**.

Предсказване на преходите.

За да може да се предвиди разклонение в програмата, процесорът използва динамичен алгоритъм за предсказване 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 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.

4.1.2. Конвейер за реални числа.

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. Правилата за сдвояване не допускат паралелно изпълнение на команди с цели числа и такива с плаваща запетая.

Само командата **FXCHG**, която разменя местата на два елемента в регистровия стек, може да бъде изпълнена едновременно с друга команда за числа с плаваща запетая.

Чрез тези основни правила се избягва тясното място, което се обуславя от стеково ориентираната регистрова архитектура на устройството с плаваща аритметика. Общо **FPU** има осем регистъра, които са изградени на принципа на стека. Зареждането на най-горния регистър в стека най-често се извършва чрез командата **FXCHG**. Ето защо тя е вградена в архитектурата на Pentium по такъв начин, че да може да се изпълнява паралелно с другите команди на аритметиката с плаваща запетая. За нейното паралелно изпълнение са необходими “квази нула такта”, а ако това не е възможно – един такт.

От направеното кратко изложение на работата на конвейерите става ясно, че разрешаването на междукомандните зависимости се реализира чрез изпълнението на изключително строги правила, заложени в управлението на конвейерите. В случая не е възможна промяна на реда на изпълнение на командите.

4.2. Архитектурни особености на конвейерите в P6.

Архитектурата P6 също е суперскаларна. Тя включва три целочислени конвейера, вместо двата на P5. Конвейерите са с повече степени, т.е. те работят с повишена честота.

Усъвършенстването на работата на конвейерите, в сравнение с архитектурата P5 се търси в следните четири направления:

- 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.

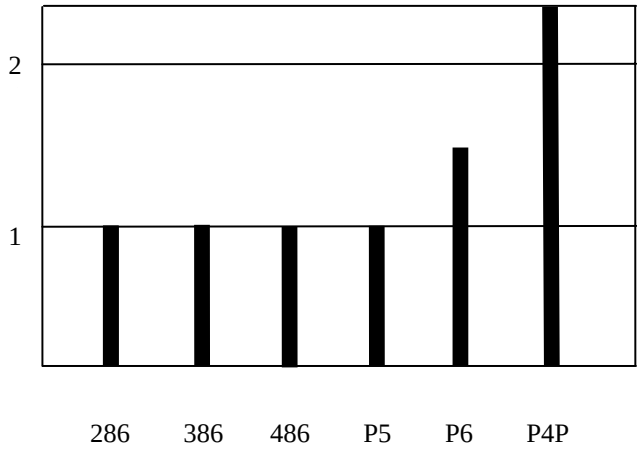
P6 цели да извлече максимума от своята производителност, като използва техника на непоследователно изпълнение на операциите, която му позволява да изтегля команди, за да поддържа конвейерите си максимално натоварени. Но някои команди не могат да бъдат изпълнявани безразборно. Тези команди правят суперконвейерната архитектура доста уязвима, което налага процесорът да забавя всички други операции, за да извърши изпълнението на една единствена специализирана команда. Сред специалните случаи спадат операциите за четене от цял регистър, които могат да бъдат забавяни от предхождащи ги операции на запис в отделен сектор на същия регистър, напр. в 8-битовата част на 16-битов регистър. Влиянието което оказват такива команди в 16 битовите програми е значително. Тези команди могат да забавят процесора за седем или повече цикъла, което означава загуба на възможността да бъдат изпълнени близо 20 команди x86. (Четене от цял регистър, следвано от запис в 8 или 16 битови регистри при 32 битовите операции може да доведе до същия проблем, но според Intel при повечето 32 битови команди тази практика се избягва.) Още по разрушително е влиянието на зареждането на сегментите и входно/изходните команди, които не само забавят издаването на следващите команди, но също така предизвикват запълване на предшествуващите ги степени в конвейера. Всички тези проблеми са характерни за повечето от съществуващите 16 битови

програми. Това обяснява защо P6 изисква работа с 32 битова операционна система и 32 приложения и само в такъв случай ще бъде по-производителен от P5.

4.3. Архитектурни особености на конвейерите в P4P

Когато P4P се появи на пазара през ноември 2000, това беше първата нова x86 архитектура на процесор на Intel от създаването на Pentium Pro. През годините преди въвеждането на P4P, архитектурата P6 доминираше на пазара в нейното въплъщение като Pentium II и Pentium III.

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

Фигура 5-13 показва, че 286, 386, 486 и Pentium (P5) процесори имат подобна конвейерна дълбочина и те биха работили на подобни тактови честоти, ако бяха изградени по една и съща технология. В архитектурата P6 приблизително се удвояват конвейерните стъпала в сравнение с по-ранните процесори и е възможно да се достигне около 1,5 пъти по-висока честота за същата процесорна архитектура. Архитектурата P4P има още по-дълбок конвейер – около 2 пъти спрямо P6.

Опасностите на дълбокия конвейер

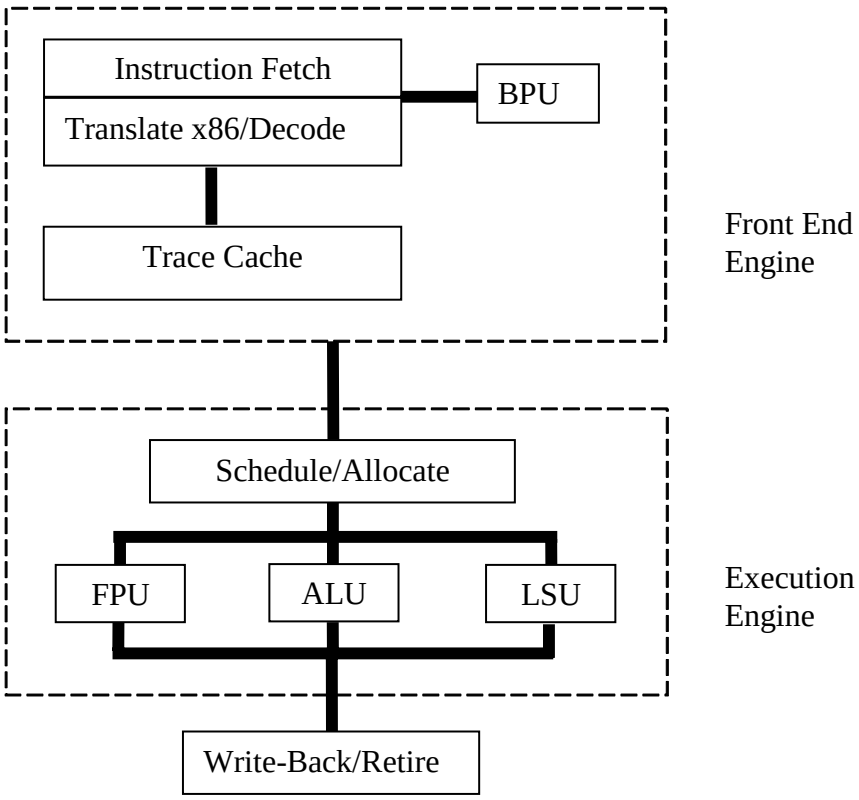
Дълбокият конвейер (общо 28 стъпала), използван в P4P, има предимство основно в 3D приложенията. Но също така той съдържа и сериозен риск, който се проявява когато не могат да се намерят достатъчно данни в L1 кеш паметта за да се захрани този дълбок конвейер. Това означава, че някои степени няма да са натоварени, т.е. появяват се така наречените мехурчета, които трябва да преминат през конвейера. Едно такова мехурче води до множество загубени цикли. Следователно, пред

проектантите на P4P е стояла задачата за предпазване появяването на мехурчета, намаляващи производителността.

За да се извлече максимална полза от дълбокия конвейер, е необходимо всички стъпала в конвейера да са пълни, следователно на стъпалото за предварителна обработка са нужни дълбоки буфери, които да могат да съхраняват и разпределят огромен брой команди (в P4P има до 126 команди в различните стъпала). Следователно, изпълнителната логика трябва да ги прегледа за зависимости, да ги преподреди (с цел отстраняване на тези зависимости) и така да ги подготви за изпълнение.

Общ поглед върху архитектурата на P4P

Обобщената структура на конвейерите в P4P е представена на фиг.5-14. На тази фигура не са показани подробно конвейерите, а е акцентувано вниманието върху взаимната връзка между отделните устройства.



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)

Както и в предходните архитектури, L1 кеш паметта е разделена на две – за команди и за данни, но тази за команди се намира в машината за предварителна обработка. Тази странно разположена кеш памет, наречена кеш памет за следите (trace cache) е едно от най-новаторските и важни особености в P4P. Да се спрем по-подробно на това разположение и да отговорим на

въпроса “Защо кеш паметта за команди е разположена на това място?”

Както стана ясно, в процесорната архитектура P6, командите x86 се разделят на по-малки, но еднообразни и по-лесно изпълними команди, наречени микрооперации (microops). Тези микрооперации са в действителност това, което изпълнителната машина преподрежда, разпределя, изпълнява и оттегля. Това добавя няколко стъпала към основния конвейер, които изискват също така и няколко конвейерни цикъла. За блок от команди, който се изпълнява еднократно, или само няколко пъти в хода на едно програмно изпълнение, тези няколко цикъла всеки път не са голям проблем. Но за блок от команди, който се изпълнява няколко хиляди пъти броят на циклите изразходвани многократно за транслация и декодиране на същата група от команди може да нарасне бързо. В P4P се спестяват тези цикли като в L1 кеш паметта се запомнят именно тези микрооперации. Кеш паметта в P4P взема транслираните, декодирани микрооперации, които са първични и готови да бъдат изпратени към машината за непоредно изпълнние и ги подрежда в малки, мини програми, наречени следи (trace).

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.

Повечето от x86 команди се декодират в 2 или 3 микрооперации, но има изключително дълги и за щастие рядко срещани x86 команди, които се декодират в стотици микрооперации (напр. командите са обработка на низ). Както и Athlon, така и P4P има ROM за микрокода на тези дълги команди. За съхраняване на тази дълга, предварително опакована последователност от микрооперации.дизайнерите на P4P са изобретили следното решение. Когато срещне една дълга команда, в сегмента на следата не се вмъква цялата последователност от микрооперации, изтеглени от тази специална ROM, а се поставя етикет (указател), който указва към секция от ROM-а, съдържащ микрокодовата последователност за тази команда. После, в изпълнителният режим когато кеш паметта за следата извежда потока от команди към изпълнителната машина и срещне един от тези етикети, тя спира и временно предава управлението на командния поток към ROM паметта съдържаща микрокода. Последната въвежда подходящата последователност

от микрооперации в командния поток и след това връща управлението обратно към кеш паметта на следата. В действителност изпълнителната машина не знае дали командите идват от кеш паметта на следата или от микрокодовата ROM.

Вероятно, това е причината, иначе малката L1 кеш памет (само 12 K) да е достатъчна да захранва дълбокия конвейер на P4P. Според Intel тази памет е приблизително еквивалента на 16-18 K "стандартна" L1 кеш за команди.

Литература

1. П.М.Коуги, Архитектура конвейерных ЭВМ, М., "Радио и связь", 1985, стр.123-151.
2. Стоян Марков.
Изчислителни системи с висока производителност, С., Техника, 1990, стр.39-64.
3. Амамия М., Танака Ю.
Архитектура ЭВМ и искустенный интеллект., М., "Мир", 1993 г., стр.57-61.
4. Р.Иванов, О.Асенов
Архитектура и системно програмиране за базирани компютри. Габрово,1998.
5. У.Стиллингс
Архитектура компьютера с сокращенным наборам команд. ТИИЭР, т.76, No 1, январь 1988р стр. 42-63.
6. Всичко за Pentium. НИСОФТ ООД,1998, 83-123.
7. Н. Рускова
Микропроцесорни системи., Печатна база при ТУ-Варна, 1999.
8. Jon "Hannibal" Stores
Understanding Pipelining and Superscalar Execution
http://arstechnica.com/paedia/c/cpu/part-2/cpu2-1.html
9. Jon "Hannibal" Stores
The Pentium 4 and the G4e: an Architectural comparison: Part I; Part II.
http://arstechnica.com/articles/paedia/cpu/p4andg4e.ars

ТЕМА 6

ПРОЦЕСОР С МНОЖЕСТВО ФУНКЦИОНАЛНИ УСТРОЙСТВА

1.Въведение

Друг начин, различен от конвейерния, за постигане на висока скорост на изчисление е да се използват множество функционални устройства (ФУ) вътре в единичния процесор. В този случай всяко ФУ се активира чрез отделна команда. ФУ могат да бъдат еднотипни по структура, т.е. *универсални* по функциониране или разнотипни по структура, т.е. *специализирани* по функциониране. Някои от ФУ могат да се повтарят. Освен това, ФУ могат да бъдат *прости* ако следващата операция не може да започне преди да е завършела предидущата и *конвейерни*, когато самото ФУ въплъщава конвейерен принцип на работа.

В отличие от векторния процесор (виж теми 4 и 7), ефективното използуване на оборудването не изисква регулярност на данните, т.е. те да са структурирани във вид на вектор на ниво потребителска програма. Следователно този тип процесори ще постигат своята максимална производителност за скаларни операции. От друга страна, за разлика от мултипроцесорната архитектура (виж теми 8 и 10), функционирането на системата не предполага съществени издръжки свързани с комуникацията между ФУ.

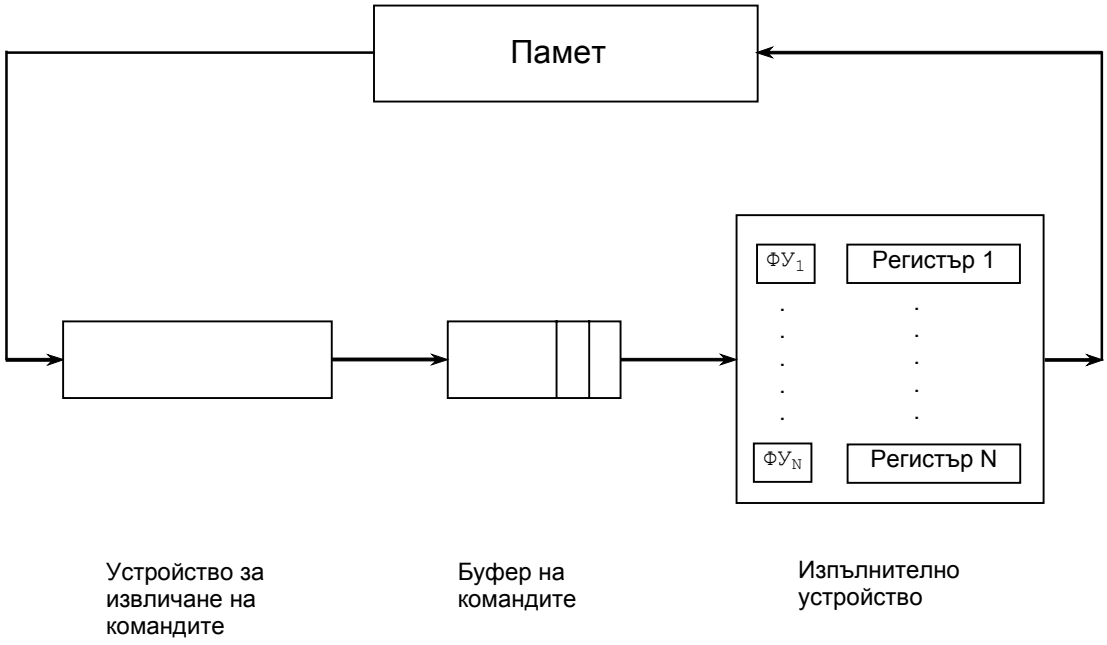
Разбира се, множеството от ФУ имат за решаване твърде важен проблем, известен като синхронизация между устройствата. Синхронизацията в конвейера се постига чрез самата структура на данните. При процесор с няколко ФУ проблемът е малко по-сложен. От една страна, различните команди могат да заемат различно време за изпълнение. От друга страна, времето за изпълнение на една команда се влияе от местоположението на операндите й, а също така и от това къде се записва резултата. И на края, някои от входните операнди на дадена команда могат да бъдат резултат от друга команда, изпълнявана по същото време, т.е. съществува зависимост по данни. И така, проблемът за синхронизацията е комплексен и неговото решение се търси по два начина: чрез апаратни средства и чрез програмни средства. И в двата случая се реализира паралелизъм на ниско ниво (Instruction Level Parallelism), при който потребителят не подозира, че неговата програма (задача) се изпълнява паралелно..

2. Синхронизация на апаратно ниво

Синхронизацията се реализира по време на изпълнение на програмата. Този подход исторически първи се е появил в компютъра CDC 6600 през 1964. Обобщена структурна схема на такъв процесор е дадена на фиг.6-1. От нея става ясно, че по същество това е конвейер за команди, в който с цел намаляване на времето за изпълнение на командите, изпълнителното устройство е съставено от

няколко паралелно работещи ФУ. Така става възможно да се реализира *функционален паралелизъм* заложен в програмата. Другото име, по-популярно, под която е позната тази архитектура е суперскаларна архитектура.

Ако се организира изчислителния процес така, че ФУ да взаимодействат само с регистрите за получаване и записване на данните, сравнително лесно се решава един от проблемите свързани със синхронизацията – зависимостта на времето за изпълнение на командата от местоположението на данните.



Фиг.6-1. Обобщена структура на процесор, реализиращ функционален паралелизъм

Почти всички съвременни процесорите, включително семейството Pentium III и процесорът P4 на Intel, K6 и K7 AMD, Alpha, PC601, PC602 на Motorola, UltraSPARC на Sun и други са суперскаларни. Разбира се ФУ в тези случаи представляват сами по себе си конвейери. Синхронизацията на паралелно работещите конвейери се осъществява чрез логика, която е значително по сложна в сравнение с тази за управление на единичния конвейер.

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. Тази ситуация е аналогична на структурен конфликт (виж тема 4). 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).

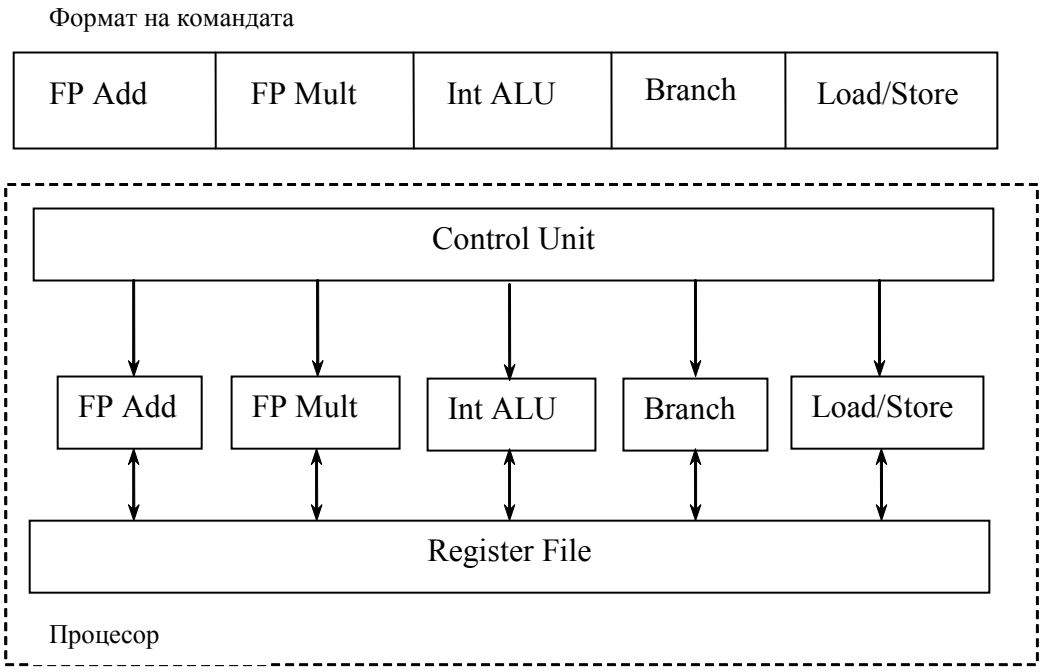
Прозорецът на изпълнение – това е набор от команди, които се явяват кандидати за изпълнение в даден момент. Всяка команда от този прозорец може да бъде изпълнена с отчитане на гореспоменатите ограничения. Количеството команди зависи от броя на конвейерите, включени в състава на процесора и броя на степените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

3. Синхронизация на програмно ниво

В този случай компилаторът е отговорен за насрочването на изпълнението на командите, т.е. за синхронизацията на множеството ФУ. Този подход води до архитектура известна под името процесор със свръхдълга команда - **VLIW** (**V**ery **L**ong **I**nstruction **W**ord). В известен смисъл **VLIW** е логическо разширение на **RISC** процесорите. При **VLIW** архитектурата, компилаторът пакетира няколко прости команди в една дълга команда. Всяко поле от тази дълга команда управлява директно едно ФУ. На фиг.6-2 е дадена обобщената структурата на един **VLIW** процесор.



Фиг.6.2 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.

Компилаторът анализира един модул или подпрограма. След изпълнение на "класическите" процедури за оптимизация,

компилаторът строи граф на изпълнимата програма на ниво отделни команди.

Използвайки евристични оценки за вероятността на преходите, компилаторът избира най-вероятния път за изпълнение на програмата. След това този път се предава на генератора на изпълнителните кодове, който го разглежда като път не съдържащ преходи. Генераторът на изпълнителните кодове опакова операциите в дълга дума, отчитайки такива фактори, като зависимост по данни, оптимално разпределение на ФУ, регистрите, системната шина, обръщенията към паметта и създава обектен код.

В системата команди на процесорът с **VLIW** архитектура има команди за сравнение с установяване на флаг в едноразряден регистров файл на прехода и команда за преход по флаг. Това позволява на компилаторът да постави команди за сравнение там където е удобно, в момент не свързан с физическото изпълнение на прехода. Преходът се изпълнява, ако указания регистър за преход съдържа единица. Тази процедура е сходна с процедурата на отложеното разклонение (виж тема 5).

Изпълнението на условни преходи в процесор с **VLIW** архитектура поражда и един специфичен проблем. В типична програма един условен преход се явява на 5-8 команди. Ако повече от 5 команди се обединяват в една дълга команда, е необходимо да се предвиди някакъв механизъм, допускащ наличие в една дълга команда на няколко прехода. В компютрите TRACE [7] е въведена схема на приоритетите. Да предположим, че два прехода следващи един след друг в изходната програма, се пакетират в една свръхдълга команда. Тогава е необходимо да се установи между тях приоритет, който определя адреса за предаване в случай, когато са изпълнени и двата прехода. Очевидно, последователността на командите в изходния текст на програмата ще определи и приоритета.

Всяка операция явно е определена на нивото на системата команди, така че компилаторът може оптимално да планира тяхното изпълнение. Конвейеризацията осигурява възможност за стартиране във всеки цикъл на нова команда във всяко ФУ. Този явен паралелизъм позволява максимално да се използува производителността на оборудването, защото функционалните устройства не се изчакват едно друго. Конвейеризацията допълнително дава възможност за повишаване на тактовата честота.

Отсъствието на взаимни блокировки в работата на ФУ позволява да се опрости управляващата логика и да се повиши бързодействието. Съществуват оценки, съгласно които взаимното блокиране на конвейера в процесора със стандартна архитектура отнема до 15% от такта.

Всеки **VLIW** процесор съдържа голям брой пътища за данните между ФУ, управлението на които се планира във фаза компилация. Така стават излишни такива “стандартни” средства за синхронизация като арбитър на шината, опашки и др.

Пример. *Даден е следният програмен фрагмент записан на езика C, 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 6-1.

Таблица 6-1.

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 използуват 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:

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

а от тук и ефективността:

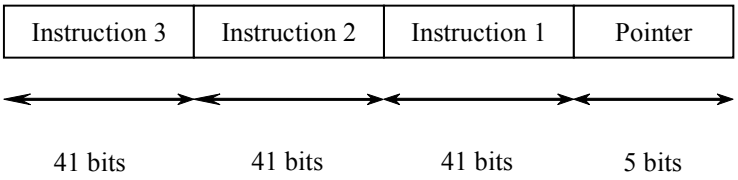
$$E = \frac{S}{P} = \frac{2.29}{6} = 0.38 .$$

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. В наши дни 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 – Efiicion 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 на кратко 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). Форматът на командата е 128 бита и е даден на фиг.6-3.



Фиг.6-3. 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 6-2 са обобщени разликите между "класическата" архитектура на Intel –x86 и новата – IA-64.

Таблица 6-2

	Архитектура x86	Архитектура IA-64
1.	Използва команди с променлива дължина, обработвани по една за "определено" време.	Използва по-прости, с фиксирана дължина команди, опаковани заедно в група от по три
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 изпълняват команди 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 и търси данните най-напред в кеш паметта.	Спекулативно зарежда данни преди тяхната необходимост и все още се прави опит за намиране на данни в кеш паметта.

4.Сравнение на двата подхода за синхронизация

И двата разгледани подхода за увеличаване на производителността, използват паралелизъм на ниво машинна команда Същевременно е налице разлика между тях; и двата подхода имат предимства и недостатъци. Едно от най важните предимства на синхронизацията реализирана на хардуерно ниво е, че произволен код, създаден за едно поколение процесори може да се използува и

на следващото поколение процесори, което може да има различен (по-добър) конвейер или различен брой ФУ. Въпреки, че този код може да се подобри при рекомпилация, това обикновено не се налага, защото подредбата **FIFO** налага проста дисциплина на обслужване, която е лесна за поддържане между поколенията.

Цената на хардуерното насрочване на управлението и присъщата му гъвкавост между поколенията е сложността. Логиката на декодиране и издаване на команди трябва да е много интелигентна, за да определи проблемите, породени от изпълнението на стар код на по-нови, суперскаларни процесори. Броят транзистори, необходими за осъществяването на такова ниво на интелигентност е значителен – свидетелство за това са сложните механизми за следене на командите използувани в PowerPC 620, K5, K6, K7 на AMD, T5 на MIPS, P5, P6 и P4P на Intel и др., а времето, необходимо за изпълнение на тази работа също се прибавя към времето за работа на конвейера. По-прости фази за декодиране и издаване на командите би позволило на тактовата честота да се увеличи, тъй като тези фази обикновено имат най-голяма латентност в суперскаларните и суперконвейерните процесори в момента.

Предимството на **VLIW** се проявява именно в последното – като се премахне сложността в хардуера, може да се създадат по-прости, но работещи на по-висока честота процесори. От една страна, по-простия хардуер позволява да се увеличи тактовата честота по-агресивно отколкото е възможно при сложните **RISC** чипове. От друга страна, лесно може да се добавят повече ФУ, за да се използува целия паралелизъм, който съществува в кода на програмата. За да работят **VLIW** процесорите добре, е необходимо наличието на интелигентни компилатори, определящи кои операции могат да се изпълняват паралелно. Това решение се взема по време на компилирането и се фиксира като командите се пакетират в дългата дума.

Като друго предимство може да се посочи, че разкриването на паралелизъм не изисква каквато и да е намеса на програмиста – не е необходимо да се реструктурира програмата за да се съгласува с архитектурата на компютъра, нито да се въведат явни директиви за съвместяване на операциите. Това съществено отличава **VLIW** архитектурата от тази на векторните процесори (виж тема 7), матричните процесори (виж тема 8) или компютрите с разпределена памет (виж тема 10).

За този начин на синхронизация в работата на няколко ФУ по-смущаващи са въпросите с адаптирането. Въпреки, че опростената декодираща електроника води до значителни подобрения в скоростта, по-простите декодери не могат да се адаптират така добре в динамичните ситуации по време на изпълнението като тези, които се срещат при командите за условен преход. Нещо още по-важно – понеже **VLIW** компилаторът трябва да знае подробностите на микроархитектурата на целевия чип, всеки код, който той генерира

ще се изпълнява добре само на него. Това означава, че при преминаването от едно поколение процесори към друго ще трябва да се прекомпилира целия код. С други думи проблемът за двоичната съвместимост остава открит.

5. TTA процесор

Крайният вариант на **RISC** процесора и процесора с множество ФУ е **TTA** (**T**ransport **T**riggered **A**rchitecture) процесора, т.е. процесор с транспортно - спускова архитектура [6]. Това е архитектура, която има само една команда – MOVE и множество ФУ и от тук другото наименование – MOVE processor. Както предполага името, процесорът работи само с преместване на данни от едно устройство в друго, като с това се задействува дадената операция. При традиционните процесори, програмите се описват чрез специфични команди, които местят данни между регистрите и ги променят в АЛУ. Например, операцията събиране обикновено зарежда два операнда и записва един резултат. Такива компютри (те включват **RISC** и **CISC** процесори) могат да бъдат наречени **OTA** (**O**peration **T**riggered **A**rchitecture). Процесорът с архитектура **TTA** разбива команда от този тип на три команди. Например, в **TTA** кодът за събиране може да изглежда така:

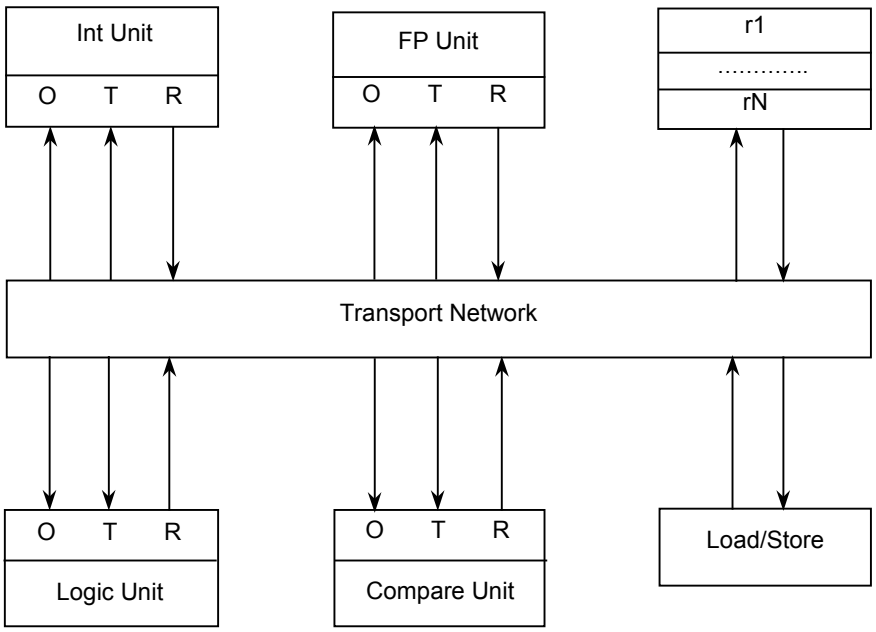
r1→add_O
r2→add_T
add_R→r3

Тука r1, r2, r3 са регистри с общо предназначение, а add_O, add_T add_R са регистри на ФУ.

Този подход може да бъде особено ефективен при наличието на голяма паралелност при изпълнение на командите. На фиг.6-4 е показана примерна структура на един такъв процесор.

Регистрите на **TTA** се делят на четири групи: регистри за операнд (**O**); регистри за задействане (**T**); регистри за резултата (**R**); и регистри с общо предназначение **r1**…**rN**. Първите три са към ФУ, имат имена и са софтуерно достъпни, за разлика от класическите процесори.

Изпълнението на двуместна операция при **TTA** процесора започва със зареждането на операнд в регистъра за операндите. Вторият операнд се зарежда в задействащия регистър, което дава знак на ФУ да започне работа. Резултатът се записва в регистъра на резултата. Този резултат трябва да бъде запазен в някой от регистрите с общо предназначение, ако трябва да се съхрани резултатът за повече от един такт.



Фиг.6-4. Вътрешна структура на един примерен **TTA** процесор

При изпълнението на командите трябва да се отбележат две неща:

- Задействащият операнд не може да се зареди преди другия операнд, но за сметка на това може да се зареди в същия такт. По принцип може да се изпълняват толкова команди **MOVE**, колкото транспортни магистрали има процесорът.
- Времето между зареждане на операнда и задействуващия операнд не може да бъде по-малко от така нареченото мъртво време на ФУ. Тука трябва да се отбележи, че всяко ФУ има собствен вътрешен конвейер и е работа на компилатора да се грижи, да изчака мъртвото време на всяко устройство.

Работата при **TTA** процесорите се следи от така наречените "пазачи", които определят дали дадена команда **MOVE** ще бъде изпълнена или не, в зависимост от един бит (също както при регистрите за състоянието на класическия процесор). Сравняващите ФУ задават стойност на този бит. Ето как ще изглежда конструкция от типа

```
if r2=r3 then goto label
```

компилирана за **TTA** процесор:

```
r2→eq_O; зарежда операнд за сравнение
r3→eq_T; втори операнд, задействащ сравнението
eq_R→b4; резултата се записва в бит b4
b4?label→pc; в зависимост от b4 преход към адрес label
```

Преходите се изпълняват като директно се зарежда нова стойност в програмния брояч, който е достъпен за софтуера при **TTA** архитектурата.

От гледна точка на проектантите, **TTA** архитектурата предлага няколко предимства, най-важното от което е голямата тактова честота и възможността да се увеличи натоварването на ФУ на процесора. Суперскаларната архитектура вече показа намаляване на ефективността си; с увеличаване на броя на обработваните команди на такт се налагат все по-сложни проверки за зависимостите между данните, а освен това нарастват и конфликтите при достъпа до ресурсите. **TTA** печели от липсата на тези задръжки. Дори още повече – отделянето на ФУ от главната транспортна мрежа означава, че всеки конвейер на ФУ може да бъде оптимизиран по отношение по по-добро време на такт. Главната транспортна мрежа може да се разглежда като суперконвейер, като в този случай долната граница на времето за такт на чипа е времето за трансфер на данни от един регистър до друг по мрежата. **TTA** процесорът много лесно се разширява (мащабира), тъй като добавянето на нови ФУ и транспорти се превръща в почти механичен процес.

Като добавка трябва да се спомене, че **TTA** прави възможни няколко нови вида оптимизации по време на компилация. При тях има много по-голяма свобода при реда на изпълнение и компилаторът може да изпълни някои от следващите команди, преди получаването на даден резултат. Преминаването през регистър с общо предназначение, което е хардуерно вградено при класическия процесор, при **TTA** процесорът е софтуерно. Резултатът от дадено ФУ може да бъде зададен директно в следващото ФУ, напр.:

add_R→mul_O
add_R→r3.

По този начин, при наличието на повече независими магистрали е възможно двете команди да се изпълняват в един такт. Освен това, ако компилаторът установи, че резултатът не се използува повече, може да не зареди стойността в регистъра с общо предназначение, като така се спестява излишен код (втората команда в примера). Същевременно, ако стойността се използува повече от един път е необходимо само първоначалното й зареждане. В резултат на тези оптимизации, **TTA** процесорът може да има по-малко регистри с общо предназначение в сравнение с **RISC** процесорите, поради това че няма да има нужда да се съхраняват всички резултати.

Какво е бъдещето на **TTA**?

Тази архитектура е предложена от Хенк Корпорал от Holland University of Delft и е част от едно изследване върху интегрални схеми със специално приложение. Независимо от това определен интерес към тази архитектура проявяват такива фирми като Intel и HP.

Опитният образец на такъв процесор е показал от 25% до 50% по-висока производителност от подобен "класически" процесор с еквивалентен брой ФУ и същата тактова честота.

Литература

1. Хокни Р., Джессхоуп К.
Параллельные ЭВМ.,М.,"Радио и связь", 1986 г., стр.130-143.
2. Мотоока Т.
Компютери на СБИС (в 2-х книгах), Мир, Москва, 1988 (1 – стр.64-66).
3. Ник Стам
Intel се стяга за новото хилядолетие. PC Magazine Bulgaria, Октомври 1999,стр. 7.
4.Том Р. Халфил
След Pentium II. BYTE BULGARIA, стр. 4-11.
5. Гордън Ланг
Все по-бързо и по-бързо., Personal Computer World, октомври 2000, стр. 43-50.
6. Архитектура, базирана на транспортирането на данни. Computer News., Извънреден брой за **RISC** технологии, 1995, стр.25-27.
7. Colwell R.P., Nix R.P., O'Donnel J.J., Papworth D.B., Rodman P.K.
A **VLIW** architecture for a trace scheduling compiler. "IEEE Transaction on Computer", 1988, V.37, No.8, pp.967-979.
8. Нови подробности за процесорите Intel Madison и Montecito. Хардуер., бр.3, март/април 2003, стр.9.
9.Мобильный процессор Tranmeta Crusoe TM5800.
http://www.iru.ru/notebook_art.php?aid=12
10. Stephen R. Wheat
Intel Itanium 2 Processor: Architecture Overview and Computing for Performance.
http://www.sharcnet.ca/fw2003/slides/StephenWheat_Intel.pdf#search='stephen%20r.%20wheat'
11. Sverre Jarp
IA-64 Architecture: A Detailed Tutorial
Http://nicewww.cern.ch/~sverre/SJ.html
12. Move Processor:Simple processor based on the TTA concept. http://www.ht-lab.com/freecores/move/move.html

ТЕМА 7

ВЕКТОРНИ ПРОЦЕСОРИ

1. Въведение

Терминът "матричен процесор" се използува в различните източници за означаване на съвършено различни архитектури. Във всеки случай става въпрос за обработка на голям обем на данни, и то структурирани във вид на масиви. Такива приложни задачи са обработката на изображения, обработката на сеизмични данни, управлението на ядрените реактори, прогнозирането на климата и т.н. и т.н.

Има принципно два подхода за изграждането на архитектурата 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. В табл.7-1 са дадени основните данни са някои от успешните векторни процесори.

Таблица 7-1

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

• Вторият подход на изграждането на архитектурата се базира на пространственото повторение на изпълнението на командата. За целта се използуват 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 представители са Staran, DAP на ICL, Exemplar 1200/CD 2 или Exemplar 1200/XA 8 на Convex, SP2 на IBM, Power Challenge на Silicon Graphics Inc.и др.

Необходимо е да се подчертае, че паралелността във времето за векторните компютри и пространствената паралелност за компютрите с архитектура **SIMD** води до съвършено различни методи и алгоритми за разработка на ефективни програми за тези два класа компютрите.

2. Принципи на векторната обработка

Векторът е набор от скаларни данни, всички от един и същ тип, съхранявани в паметта. Обикновено векторните елементи са подредени, така че да имат фиксирано адресно нарастване между следващите елементи, наречено стъпка.

Векторна обработка се среща когато аритметични или логически операции са приложени към векторите. Преобразуването от скаларен към векторен код се нарича *векторизация*, а съответните компилатори се наричат *векторизиращи компилатори*.

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
```

т.е. за векторният процесор същата задача се решава с една команда. Тази една команда представя многото команди от скаларния процесор; така не само може да се прескочат всичките адресни декодирания, но в крайна сметка има само една команда за декодиране.

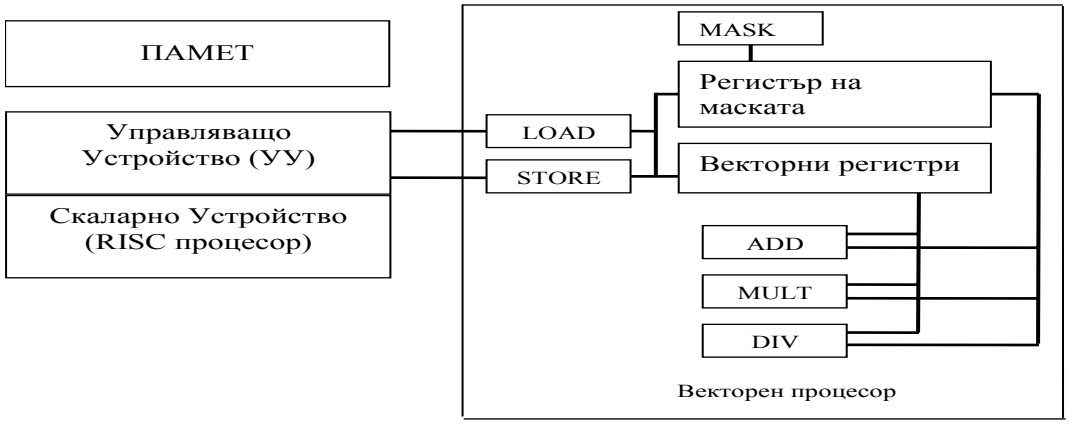
3. Структура на векторния процесор

Векторният процесор е набор от следните хардуерни ресурси: векторни регистри, функционални конвейери, обработващи елементи и броячи за изпълнение на векторни операции.

Преди да се разгледа структурата на векторния процесор, трябва да се подчертае, че съществуват два типа векторни процесори:

- Пълен векторен процесор. В този случай е по-правилно да се говори за векторен компютър, защото става въпрос за едно интегрирано устройство (компютър) предназначено за обработка както на вектори, така и на скалари.
- Присъединен векторен процесор. Този процесор представлява специализирано устройство, което може да се включва в състава на скаларният компютър, като по този начин разширява, по-точно увеличава, неговите възможности за обработка на вектори и матрици. Връзката между присъединения векторен процесор и скаларния компютър се реализира чрез канал. Така, от гледна точка на скаларния компютър, векторният процесор се разглежда като периферно устройство. Това обуславя по-ниската скорост на обмен на данните между двете устройства и води до по-ниска производителност в сравнение с първия тип векторни процесори. Ясно е, че това се явява недостатък на този тип векторни процесори, но от друга страна цената им е значително по-ниска и което не е за пренебрегване, тяхното включване към скаларния компютър може да стане когато има нужда от ефективно решаване на задачи с матрици.

Една обобщаваща структура на векторните процесори е показана на фиг.7-1.



Фиг.7-1. Обобщена структура на примерен векторен процесор. (ADD, MULT, DIV, MASK, LOAD и STORE са специализирани функционални устройства, работещи на конвейерен принцип)

Основните компоненти на векторния процесор се явяват векторните регистри, всеки от които трябва да има два входа за четене и един вход за запис; векторни функционални устройства, които могат да стартират векторна операция на всеки такт; векторни устройства LOAD/STORE за прехвърляне на векторни думи между регистрите и паметта и обратно. Чрез векторните регистри се реализира вътрешна памет и се осъществява бърз достъп до данните. Те представляват няколко регистъра с фиксирана дължина и размери от 128 до 256 KB, с дължина 64-2048 думи (като правило думата е с дължина 64 бита).

Функционалното устройство MASK и регистърът на маската обработват специален вектор на маската (двоичен вектор), който се използва при адресация на разредени матрици (вектори). Работата с тези вектори ще бъде разгледана по-долу.

Оперативната памет е сходна по функции с оперативната памет на скаларните компютри. Но изискванията към нея по отношение на времето за достъп е по-високо в сравнение със скаларните компютри. В разгледания по-горе пример, паметта трябва да доставя един елемент от A и B за всеки такт. АЛУ (конвейерът за събиране) също произвежда един резултат за един такт. Трудността в проектирането на паметта е да се поддържа непрекъснат поток от данни от паметта към АЛУ и обратно. Следователно за C=A+B системната памет трябва да има най-малко три пъти по-широка лента на пропускане в сравнение със скаларния процесор. Основните стратегии за осигуряване на това изискване са:

- 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 13).
- 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.

4 Векторни команди

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

- Код на операцията т.е. функция, която се изпълнява.

• Операндите, които се използуват.

• Състоянието след изпълнението на командата, което трябва да бъде запомнено (фиксирано).

• Адресът на следващата команда, която се изпълнява.

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

4.1. Код на операциията.

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.

Конкретната структура на съставните векторни команди зависи от самия векторен процесор. За апаратна поддръжка на съставната команда е необходимо да работят паралелно и/или последователно няколко аритметични конвейери Тяхното преимущество се състои в това, че при прекъсване се съхранява информация за една команда, а не за няколко паралелно изпълнявани команди.

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.3.3.

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 използува в компютрите на CDC Cyber 200/205.

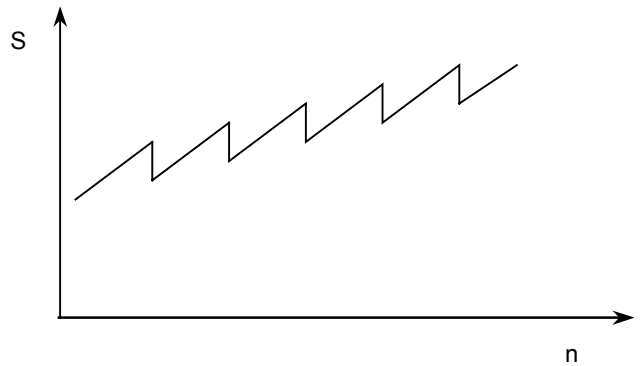
• Адреси само на регистри, т.е. адресация от тип регистър-регистър. Такава адресация напр. се използува в компютрите на Cray Res.

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

Нека да посочим какви са предимствата на всеки отделен тип адресация [3].

Командите от тип "памет-памет" изискват най-голямо време за обработка, но за сметка на това са инвариантни по отношение на дължината на векторите. Точно в това се проявява тяхното най-голямо предимство. Освен това в набора команди на векторния процесор липсват команди за четене/запис от/в регистрите.

Командите от тип "регистър-регистър" са най-бързи, но използуването им налага известни ограничения както върху архитектурата, така и върху програмирането. При този тип адресация задължително трябва да присъствуват и команди за четене/запис от/в регистрите. Крайната дължина на регистрите създава известни проблеми при осъществяване на тези операции. Нека да разгледаме този въпрос малко по-подробно. Да предположим, че векторните регистри са с дължина 64 елемента, а обработвания вектор е с дължина 136 елемента. За да се обработят всичките елементи на вектора е необходимо три обръщения към паметта за зареждане на регистрите и още три обръщения за да се прехвърлят данните в паметта. И в двата случая третото обръщение към регистрите няма да използва цялата дължина на регистрите (само 8 елемента от тях), но ще отнеме същото време като предходните две. Това обуславя специфичния "трионообразен" характер на изменение на производителността **S** от дължината на вектора **n** – фиг. 7-2.



Фиг.7-2. Типична зависимост на производителността на векторен процесор от дължината на вектора при адресация от типа регистър-регистър.

В действителност, адресацията памет-памет може да се окаже по-ефективна, ако векторите са достатъчно дълги. Въпреки всичко, експериментите показаха, че късите вектори са по-често срещани.

Като един добър компромисен вариант, съчетаващ предимствата на двата предходни способа за адресация може да се посочи адресация от тип "памет-регистър". Този тип адресация изисква най-малък брой тактове за изпълнението им. Това особено важи за съставните команди.

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.

Първите четири позиции имат тривиален начин за задаване, затова по-надолу ще се спрем само на последната.

По цял ред причини, елементите на вектора не винаги попадат в съседни клетки на паметта. Например, обръщенията към стълб на матрица, елементите на която се съхраняват по редове. Затова разположението на данните в паметта придобива голямо значение.

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.

Плътна (регулярната) схема за адресация има от своя страна три разновидности:

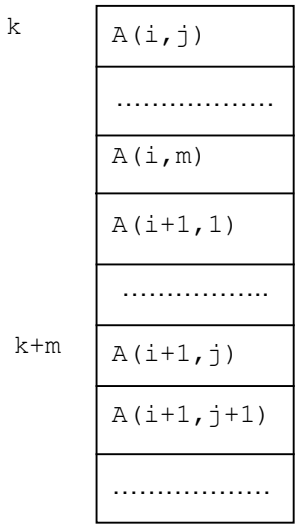
- последователна;
- непоследователна, но регулярна;
- подматрична.

Нека да разгледаме тези адресации.

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.7-3 е дадено разположение на елементите от редовете на **A** в последователни клетки от паметта.



Фиг.7-3. Разположение на елементите матрицата **A** в паметта

За да се получи достъп до вектора-стълб по тази схема, генераторът на векторни адреси трябва да може да генерира серия от адреси, значението на които се отличава на **m**. Понеже **m** се мени, то неговата стойност трябва да бъде включена по някакъв начин в полето на командата.

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:

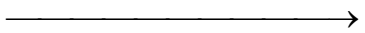
- Операциите зависят от данните.
- Обработват се разредени вектори.
- Косвено се преглеждат таблици.

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.

Двоичен вектор.

Двоичният вектор е вектор с дължина равна на дължината на обработвания вектор, но неговите елементи приемат стойност логическа единица или логическа нула. Двоичният вектор се явява маска, която се налага върху обработвания вектор. Ако i-тият бит на двоичния вектор е 1, то i-тият елемент от обработвания вектор се подлага на по-нататъшна обработка. На фиг.7-4 е показана функцията свиване.

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	



Свиване

Фиг.7-4. Функцията “свиване” при работа с вектори

Функцията разширение е обратна на свиване. Разбира се , ако се изходи от вектор с елементи 1.1, 4.4, 5.5, 8.8, стойностите 2.2, 3.3 и т.н не могат да се възстановят. В зависимост от операционната система на техните места ще има нули или ще останат стойностите на клетките от паметта от предходните им използвания. При функцията “селективно запомняне”, i-тия изчисляем елемент се запомня само ако i-тия бит на двоичния вектор съдържа 1, в противен случай записването се блокира.

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. Фиксиране на състоянието 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 използуват кардинално различни способи. Предложени са следните способа (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.

Преди да се изложи същността на последния, трети способ, нека да разгледаме какви проблеми възникват с прилагането на предходните два.

Даже простата фиксация на особеното състояние създава във векторния процесор проблеми с организацията на работата. Разширеният скаларен код на признаците може да покаже има ли особено състояние и къде е станало първото от тях, но не показва колко са всичките и къде са станали останалите. Векторът с кода на признаците може да показва къде именно са станали особените състояния в много векторни команди, но той е сравнително безполезен, когато всеки резултат зависи от няколко предходни операции или даже от всичките данни. Ако все пак се реализира, то на програмиста е изключително трудно да проследи пътя от особеното състояние назад до входните данни. Затова голяма част от векторните архитектури в такива случаи или прекратяват изпълнението на командите или правят някакво автоматично изправяне и продължават изпълнението, като на програмиста се посочва най-големия, разширен код на признаците.

- 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.

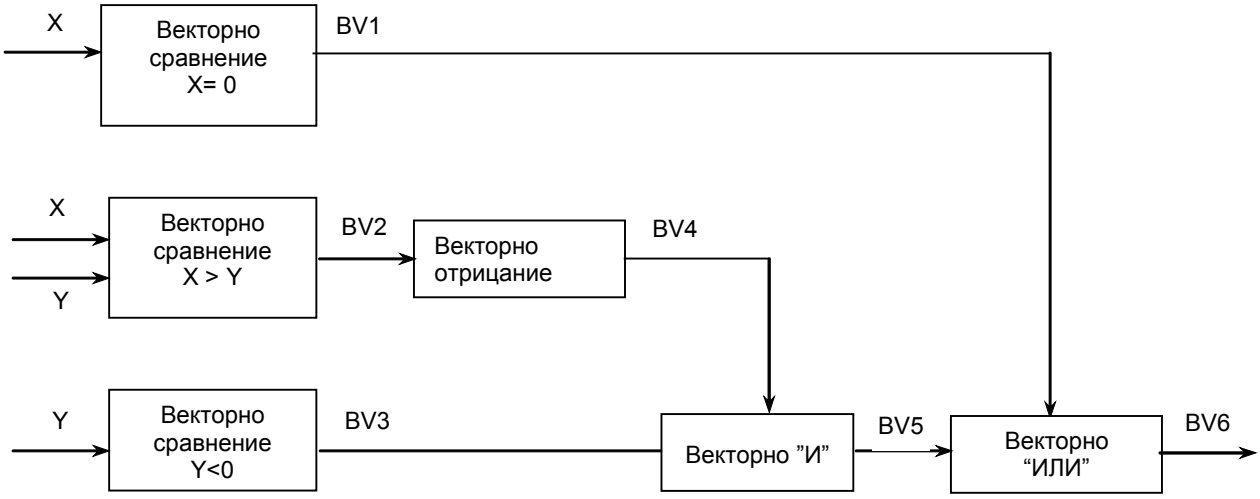
Това именно е третата причина, която води до включването на нови команди и в крайна сметка за увеличен общ брой векторни команди на процесорите (виж т.3.1).

5. Програмиране на векторния процесор.

На края, обсъждането на векторните процесори ще завърши с разглеждането на един пример, чиято цел е да покаже както използуването на двоичните вектори, а така също и по-различния начин за програмиране на векторните процесори [1].

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

Всеки правоъгълник представлява една векторна команда. Всяка една от тях приема входните вектори и изработва като резултат един двоичен вектор **BV1…..BV6.** В хоризонтално направление са разположени командите, изпълнявани последователно, а във вертикално – паралелно, ако за това има възможност. Векторът **BV6** съдържа крайния резултат, т.е. той посочва позициите на тези елементи от вектора **X**, които удовлетворяват поставените изисквания.

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 7-1 заедно със стойности на входните векторите **X** и **Y.**

Таблица 7-1

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

Литерарура

1. П.М.Коуги.
Архитектура конвейерных ЭВМ, М., “Радио и связь”, 1985, стр.151-192.
1. Хокни Р., Джессхоуп К.
Параллельные ЭВМ.,М.,”Радио и связь”, 1986 г., стр.73-145.
2. Padegs A., Moore B.B, Smith R.M., Bucholz W.
The IBM 370 System/370 vector architecture: design consideration. “IEEE Trans. Computers”, 1988, Vol.37, No5, pp.509-520.
3. Dongarra J.J., Hinds A.
Comparison of the CRAY X-MP-4, Fujitsu VP-200 and Hitachi S-810/20. “Simulation”, 1986, Vol.47, No3, pp. 93-107.
4. Gustafson J. Heinrich M.
Memory-mapped VLSI and dynamic interleave improve performance. “Computer Design”, 1985, Vol.24, No 15, pp. 15,93-96-98-100.
5. L.D.Fosdick, C.J.C.Schauble and E.R.>Jessup
General Architecture of Vector Processors. www.VectorComputerTutorial-General Architecture.html

ТЕМА 8

МАТРИЧНИ ПРОЦЕСОРИ (SIMD ПРОЦЕСОРИ)

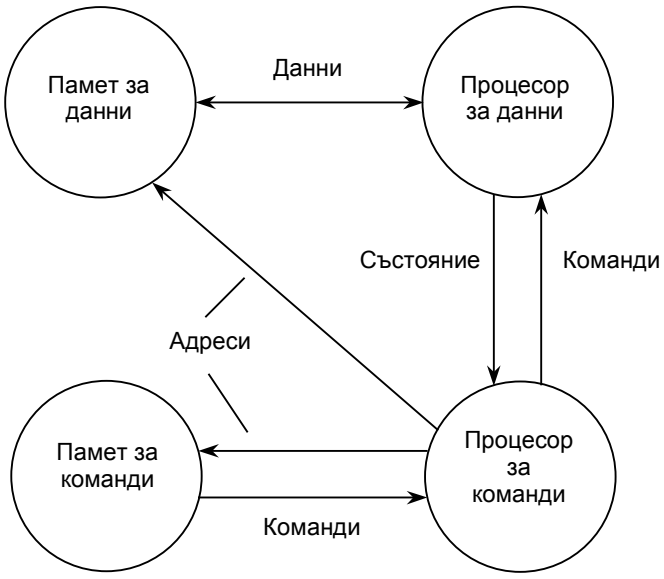
1. Въведение

В тема 7 бяха разгледани особеностите на векторните процесори, реализиращи един от подходите за изграждане на високопроизводителни компютри за матрични изчисления. Както е известно, съществува и втори подход базиран на пространствения паралелизъм. При този тип процесори, наричани **матрични процесори**, множеството процесори са свързани по подходящ начин, най-често във вид на матрица.

Матричният процесор е синхронен паралелен компютър който се състои от множество процесори, управлявани от едно единствено управляващо устройство. То извлича и декодира командите от паметта, размножава управляващите сигнали към всички процесори в матрицата, така че всички процесори изпълняват една и съща операция в едно и също време.

В зависимост от режима на работа на процесорният елемент и получаваните сигнали за управление, матричните процесори могат да се класифицират като традиционни процесори с един поток команди и множество потоци от данни (**SIMD**), систолични процесори (виж тема 8) и асоциативни процесори.

Според някои изследователи [1], **SIMD** компютрите могат да се разглеждат като фон Ноймановски по отношение на устройството за управление. Това очевидно е справедливо, ако се счита, че устройствата за управление и изпълнение са самостоятелни единици,

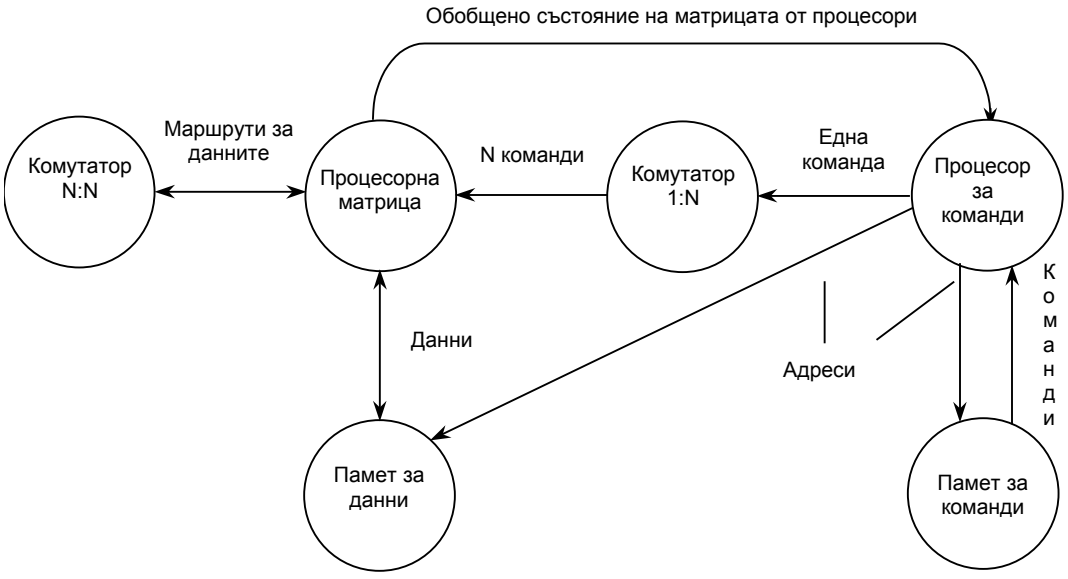


Фиг.8-1. Модел на работата фон Нойманoвски процесор

а не елементи на едно устройство - централния процесор. От тази позиция управлението на множеството процесори, свързани във вид на матрица, се оказва не отличимо от управлението на един процесор. Именно това опростява значително процеса на програмиране в сравнение със сложните процедури при другите многопроцесорни конфигурации (виж теми 9 и 10). За да се покаже по-ясно преходът от класическия процесор към матричния процесор, нека да разгледаме обобщения модел на работата на процесорът на фон Нойман, представен на фиг.8-1.

Процесорът за команди (ПК) указва адреса на данните, а така също и адреса на получените резултати. След изпълнението на командата от процесора за данни, последният връща неговото състояние в ПК. Схемата илюстрира съществуването на отделни памети за команди и данни. Това е справедливо относно кеш паметта от първо ниво; в оперативната памет командите и данните се съхраняват в отделни (обикновено изолирани) области на общата памет, докато в матричните процесори тази памет е разделена.

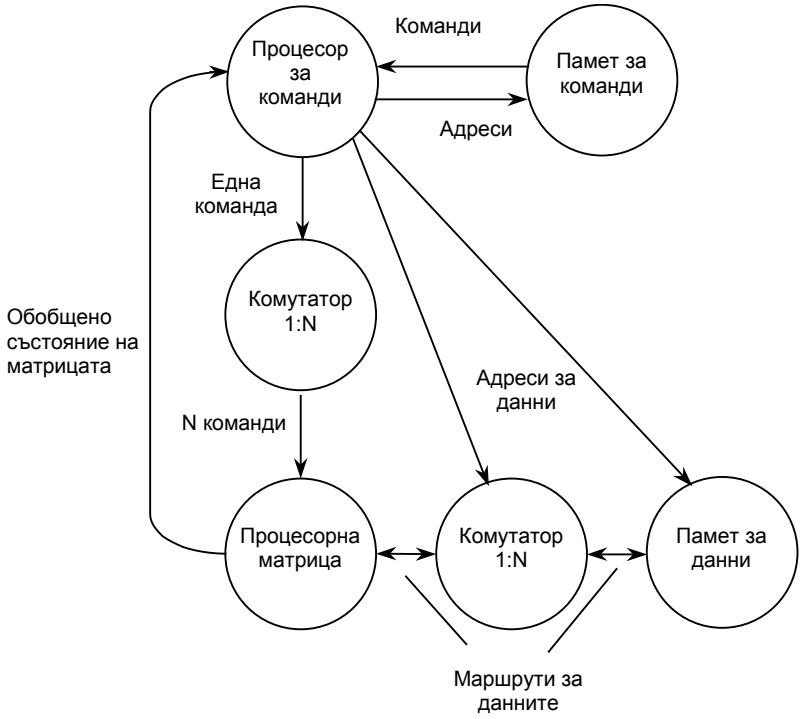
На фиг.8-2 и фиг.8-3 са показани двата възможни начина за пренос на концепцията за управление на фон Нойман върху матричния процесор. И при двата начина изпълняваната команда се предава на всички процесори за данни и се изпълнява паралелно над напълно определени, но независими данни. Комутаторът от тип **1:N**, който може да бъде установен с помощта на предварително зададено условие или специална подпрограма, осигурява размножаване на командата и последващото й изпълнение от всичките процесори или на част от тях. Двата типа матрични процесори се различават помежду си по начина на междупроцесорните съединения.



Фиг.8-2. Първи модел на матричен процесор, в който връзките между процесорите се реализират чрез комутатор от типа N:N

На фиг.8-2 е показан първият тип матричен процесор с комутатор от типа **N:N**, осигуряващ обмен на данните между процесорите. Ще отбележим, че при този тип **SIMD** компютър, обменът на данните се осъществява на ниво процесори и той се реализира по битове. Пример за **NxN** комутация може да служи обменът на данни с най-близкия съсед и е реализиран в компютъра MPP, както и в компютъра на Connection Machine, при който връзките се определят от функция на хиперкуба (виж тема 12).

На фиг.8-3 е показан матричен процесор с конфигурация от втория тип, използуваща комутатор от тип **1:N**, който осигурява прехвърлянето на данните между процесорите или между процесорите и паметта. Обменът тука е по блокове. Примери за процесори от този тип са STARAN, ASPRO и др.



Фиг.8-3.Втори модел на матричен процесор, в който връзките между процесорите се реализират чрез комутатор от типа 1:N

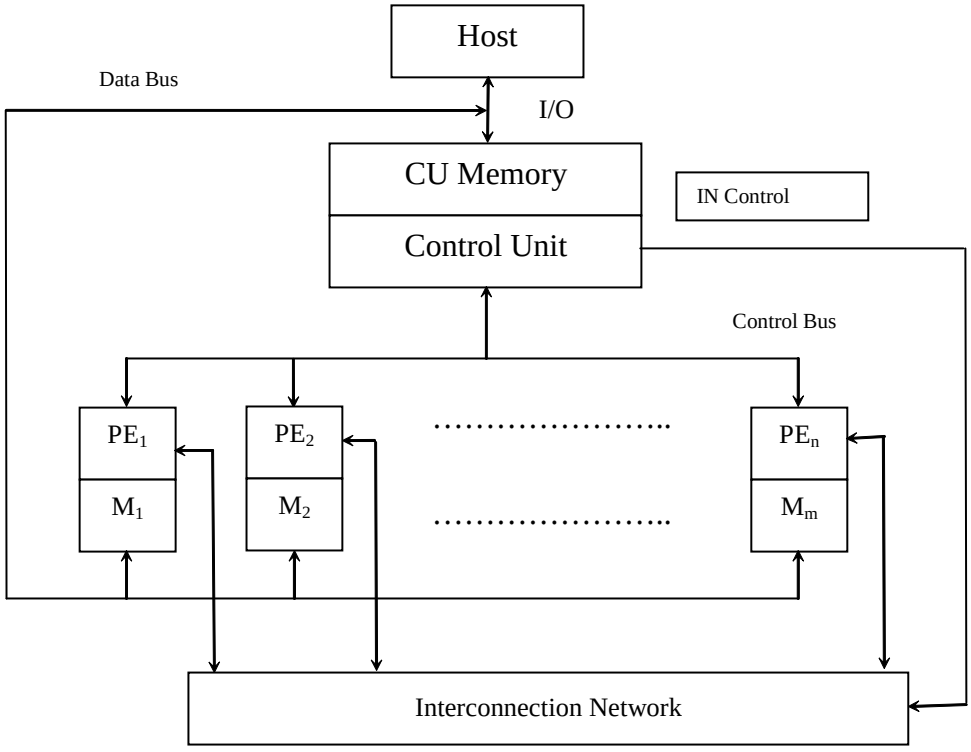
И в двата модела, командата обработена на даден етап от процесора за команди преминава през комутатор от тип 1:N, за да се размножи на N команди, толкова колкото е броят на процесорите.

2. Структура на SIMD компютрите

На разгледаните в предходната точка два начина на миграция от последователния към паралелния процесор от тип **SIMD**, съответстват и две организации на матрични процесори – с

разпределена (локална) памет и разделяема (обща) памет. Тези структури са показани съответно на фиг.8-4 и 8-5. Всяка една от тях включва матрица от процесорни елементи (PE_i), памет (M_i), управляващо устройство (CU) и комуникационна мрежа (Interconnection Network). Такива процесори са присъединени към главен (Host) компютър, който е последователен и от гледна точка на програмиста се явява машина за предварителна обработка (front-end machine). Ролята на този компютър е да изпълни компилацията на програмата, да зареди програмата, да изпълнява входно/изходни операции, а така също и други функции на операционната система.

В голяма част от задачите се срещат както скаларни данни (аргументи и параметри) така и паралелни данни (вектори, матрици, таблични данни – от всякакъв вид, организирани като файлове, съдържащи елементи с еднакъв формат). Естествено, данните от скаларен тип се обработват от последователният процесор, а тези от паралелен тип – от матричният процесор. Обобщено, последователната част на **SIMD** компютъра управлява целия компютър, защото напълно съдържа програмата и съгласува изпълнението на скаларните и матричните изчисления. При някои реализации последователната част на компютъра е стандартно произвеждан процесор

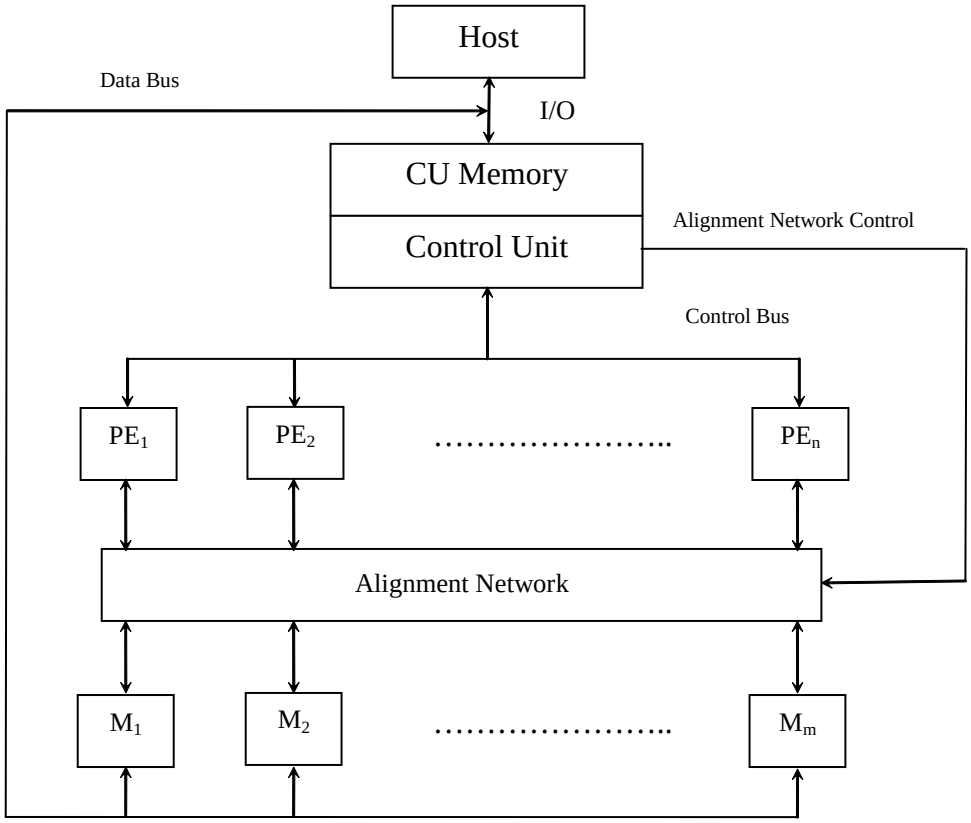


Фиг.8-4. **SIMD** компютър с локална памет

Специфична особеност на компютъра, показан на фиг.8-4 е че всеки PE има своя собствена памет. Управляващото устройство –CU извлича командите от собствената си памет (CU Memory). То ги изпълнява ако те са команди за управление или ако те имат скаларни операнди. Ако те са векторни команди, управляващото устройство ги предава на всички PE от процесорната матрица. Също така то предава и данните ако те са необходими за векторните операции. В случая, в които всеки PE се нуждае да извлече данни от своите собствени памети, управляващото устройство прехвърля адресите към PE. Нормално тези адреси са едни и същи. Обаче, за да се увеличат адресните възможности на процесора, може да се използва индексен регистър за модифициране адреса предаден от контролера. Програмистът може да установи индексния регистър във всеки процесор. Така схемата за адресиране става по-гъвкава. Комуникацията на данните е реализирана чрез преместването им от един процесор към друг чрез комуникационната мрежа. Всеки процесор, или група от процесори, има комуникационен порт и буфери за данни. Комуникационната мрежа (виж тема 12) изпълнява определен брой функции на съответствието, зависещи от топологията и управлението, получено от CU.

Броят на процесорите и тяхната разрядност, размера на локалната памет и вида на комуникационната мрежа варират от едно приложение към друго. Един от най-ранните паралелни компютри, използвал този модел, е ILLIAC IV, разработка на университета Илинойс, САЩ [6,7]. Натрупаният опит с този компютър има значително влияние върху паралелната обработка и особено върху матричните процесори. ILLIAC IV има четири квадранта, всеки от който включва 8x8 процесора, свързани по специфичен начин (комуникационна мрежа ILLIAC IV – виж тема 12). Друг компютър, използващ същия модел е Connection Machine (CM) [8]. Той съдържа общо 65536 еднобитови процесора, групиране по 16 във възел, като всичките 4096 възела са свързани в хиперкуб.

В моделът с разделяема памет, показан на фиг.8-5, паметите са отделени от процесорните елементи чрез комуникационна мрежа (Alignment Network). Всеки процесор може да има достъп до всяка памет и мрежата е такава, че тя позволява паралелен достъп към споделената памет. Предимствата на този модел са, че повече известни (разработени) алгоритми могат по-лесно да се настроят към хардуера. За да се постигне това, повече данни трябва да се превключват, което води до увеличаване на времето за достъп.



Фиг.8-5. **SIMD** компютър с разделяема памет

3. Особености на SIMD компютрите

3.1. Команди

Наборът команди за матричния процесор се разделя на три подмножества:

- Подмножество на командите за скаларната част.
- Подмножество на командите за паралелната част.
- Подмножество за управление на потока данни и команди между двете части.

Първото подмножество съдържа обикновени команди за последователен процесор. Второто подмножество не съдържа команди за преход, способни да внесат изменение в порядъка на изпълнение на командите. Устройството за последователно управление извлича команда, определя нейната принадлежност към едно от двете множества команди и в случай на последователна команда я изпълнява. В случай, че командата е за паралелната част, последователното устройство я предава към всички процесори включени в матричния процесор, които я изпълняват паралелно.

Множеството команди от третия тип управлява обмена на данните между скаларната и паралелната част на процесора. То съдържа команди изменящи потока на издаване на команди в последователната част в зависимост от резултатите, получени в

паралелната част. В него също така се съдържат команди, позволяващи да се отделят отделни процесори от матричния процесор и да се пренесе тяхното съдържание в последователната част. По такъв начин, понеже има много процесори и един поток команди, то във всеки момент от време е възможен обмен само на един процесор с устройството за управление. От друга страна, с помощта на една последователна команда могат да се предадат общи данни към всички процесори паралелно.

В **SIMD** компютрите, при паралелна обработка на големи обеми от данни много вероятна е ситуацията, когато не над всички данни трябва да се изпълняват еднакви операции. Например, ако е необходимо да се изпълни конструкцията if a_i>0 then …,очевидно само за част от процесорите ще е изпълнено условието и само те трябва да изпълняват оператора следващ then. За да се реализира тази възможност е необходимо да се въведе механизъм за блокиране (деблокиране) изпълнението на всяка команда. Този механизъм е известен под името механизъм на маската. За първи път механизмът на маската е използван в компютъра ILLIAC-IV [3]. В най-прост вид механизмът на маската се реализира чрез така наречения контекстен регистър. Ако съдържанието на този регистър е 1, то командата се изпълнява, а ако е 0, то командата се игнорира. Самото съдържание на регистъра се установява в зависимост от локалните данни. В своя най-развит вариант механизмът на маската се реализира чрез специален процесор, опериращ с булеви променливи. Входната информация за този процесор постъпва от:

- процесорите, когато в тях се изпълняват операции за сравнение (признаци от типа = , > , < и т.н.) или при изпълнението на обикновените аритметични операции (признаци от типа “резултата е равен на нула”; “резултата е по-голям от нула”; “препълване” и т.н.);
- паметта.

Под управлението на последователната част, специализираният процесор извършва логически операции над тези данни. Резултатът (също булев масив) или се записва в паметта за следващо използване, или се подава на специализиран вход на процесорите за блокиране (деблокиране) на командата.

Независимо от конкретния способ на реализация, механизмът на маската дава възможност да се въведе операционна автономия в компютъра (виж тема 3, т.5).

3.2.Синхронни операции

Наличието на едно единствено устройство за управление определя синхронната работа на матричния процесор. Исторически погледнато, **SIMD** компютрите са се появили и развили първи в света на паралелните компютри, за което съществуват две основни причини:

- да се построи изчислителна система съставена от N процесора и управлявана от едно управляващо устройство е по-евтино, отколкото система съставена от **N** процесора и **N** управляващи устройства;
- матричният процесор, работещ в синхронен режим, може по-пълно да използва лентата на пропускане на комуникационната мрежа.

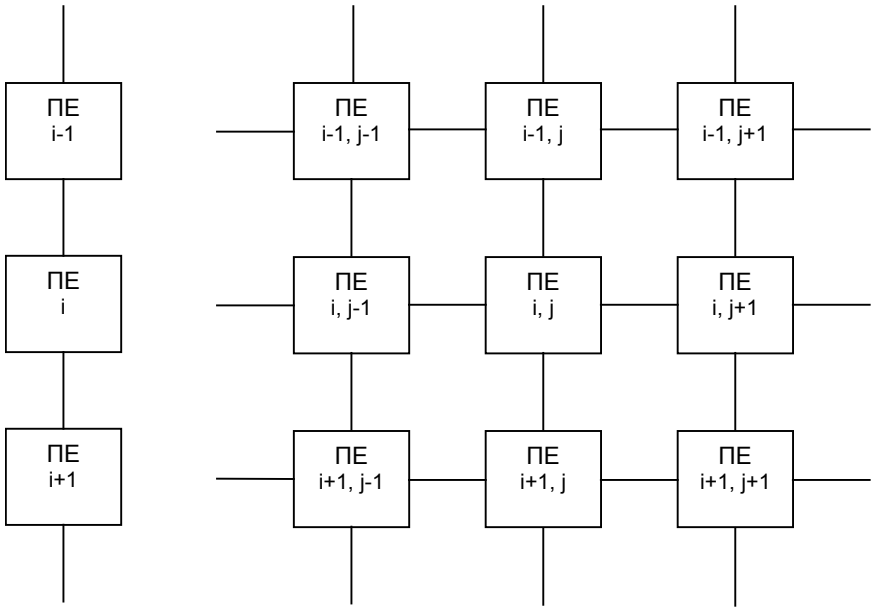
С масовото производство на милиони и милиони процесори в един чип, първата причина губи своето значение, но втората все още има своята стойност.

Разпаралелването в матричния процесор става на микониво, вследствие на което за неговата работа е необходимо щателно организирано управление. Така интерфейсът между последователното управление и матричния процесор трябва да бъде синхронен в крайна сметка до ниво операции от второто подмножество. Това означава, че след извличането на командата от управляващото устройство и предаването ѝ на матричния процесор, то спира и може да продължи работа само след получаване на отговор. На практика в някои разработки се осигурява буферизация между последователното управление и операциите на матричния процесор, така че след предаването на командата на матричния процесор, последователното устройство за управление да не стои, а да изчислява адреса и параметрите на следващата команда. Но възможностите на такъв паралелен режим на работа са ограничени, защото на устройството за управление обикновено трябват значения на някой параметър получаван от матричния процесор и за получаването му е необходимо да се дочака завършването на предидущата команда.

Вътре в матричния процесор трябва да има строга постъпкова синхронизация. Матричният контролер предава сигналите за синхронизация на всички процесори паралелно. Това ниво на синхронизация се използува за осигуряване на висока скорост на междупроцесорна комуникация и обмен на данните. Известни са различни методи за организация на междупроцесорен обмен. Например, на фиг.8-6 са показани едни от най-често използваните свързвания между процесорите вътре в матричния процесор.

Благодарение на паралелността и високата степен на синхронизация, обменът на данни в такива комуникационни мрежи се оказва изключително ефективен. Например, при получаване на команда за предаване на данни наляво (фиг.8-6б), всички процесори предават данни наляво и същевременно получават данни от дясно. Така, че общата ширина на лентата на пропускане е равна на ширината на един процесор умножена по общия брой процесори в паралелния компютър.

Регулярността и синхроността на комуникационните мрежи означава отсъствие на непроизводителни разходи за връзка, защото не трябва да се изчислява пътя в мрежата и адреса на приемащия процесор. Процесът на комуникация се свежда до три операции:



Фиг.8-6. Най-често използвани съединения между процесорите, вътре в матричния процесор.
а) топология "линийка" или "конвейер" б) топология "решетка"

- извличане на данните от паметта;
- предаване на данните към съседния процесор (с едновременно получаване на данните от съседния процесор от противоположната страна);
- запис на новополучените данни в паметта.

Разбира се, високи скорости на обмен са възможни само между съседни процесори. За предаване на данни между отдалечени процесори е необходимо повече време.

Съществува още един режим за междупроцесорна комуникация, известен като циркулационно предаване през последователния контролер. Неговата същност е следната: ако е необходимо да се прехвърлят стойности на параметри от един процесор на голям брой други процесори, или на няколко случайно разпределени и достатъчно отдалечени процесори, то понякога е по-бързо да се прехвърлят на последователния контролер и от там към всичките процесори. Чрез използване механизмът на маската може да се селектират процесорите, които получават данните.

3.3. Масов паралелизъм

Под този термин се разбира броят на паралелно работещите процесори. Ограничаващ фактор за масовия паралелизъм се явява техническата възможност за разместване на максимален брой

процесори, елементи на паметта и междуелементните съединения в едно устройство. Естествено, с развитието на технологията, се увеличава броя на паралелно работещите процесори. Така например, в началото на 70 години максималния брой процесори е 512-1024, докато 25 г. по-късно той достига до 256К. Броят на максимално работещите процесори се влияние и от предназначението на матричния процесор. Ако той е стационарен, броят на процесорите ще бъде на порядък или два по-голям отколкото случая при матричен процесор предназначен за мобилни цели, където други фактори, като малкото тегло, ниска консумация на енергия, малкия обем и т.н. са от първостепенно значение.

В много от **SIMD** компютрите, напр. DAP на ICL, в серията Connection Machine на Thinking Machine се използват еднобитови процесори. Естествено, разрядността на паметта и на шината процесорен елемент - памет е също един бит. Това означава, че изчисленията чрез ПЕ трябва да се изпълняват побитово. Коя е причината за интереса към този нестандартен начин за организация на изчислителния процес? Причините са две, а именно:

• Осигурява се голям брой паралелно работещи процесори, но с по-проста структура, които лесно биха се поместели в един чип.

• С използването на еднобитови процесори е възможно да се постигне увеличаване на скоростта на обработка.

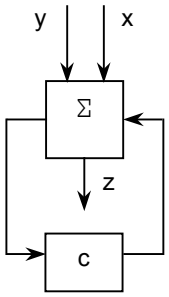
На базата на разгледания по-долу пример ще се демонстрира увеличената скорост на обработка при използването на еднобитовите процесори.

***Пример**. Дадени са два вектора – **x** и **y** с дължина **n** елемента. Всеки елемент представлява **b** разрядно число **(b ≤ n)**. Да се разработи устройство, на базата на еднобитов пълен суматор, което намира елементите на вектора **z** по формулата*

$$z_i=x_i+y_i, \quad i=1,2,\ldots n.$$

Задачата може да се реши по три начина. Преди да се разгледат по отделно трите начина, ще посочим, че времето за работа на суматора е τ_A , а времето за достъп до паметта е τ_M .

Първи начин. Първото устройство е изградено на базата на един единствен еднобитов пълен суматор – фиг.8-7. Това е възможно най-простото устройство за сумиране на два вектора с дължина **n** елемента, като всеки елемент съдържа **b** бита.

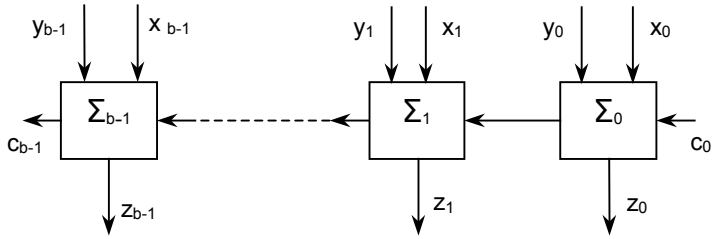


Фиг.8-7. Еднобитов суматор в качеството на градивен елемент за сумиране на два вектора с произволна дължина.

Най-напред се подават на входовете на суматора последователно битовете на първите елементи, после на вторите елементи и т.н. Изчислителният процес е строго последователен и времето $\mathbf{T_1}$ за получаване на крайния резултат е

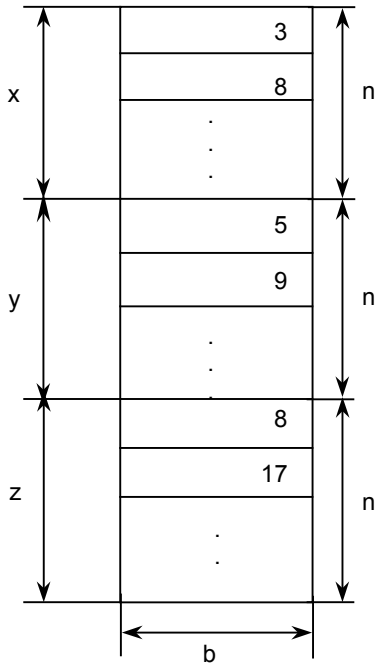
$$T_1 = nb(\tau_A + \tau_M) .$$

Втори начин. На разположение са **b** на брой еднобитови суматора, свързани по начин показани на фиг.8-8. В това устройство паралелно се обработват битовете на елементите на векторите **x** и **y**, а отделните елементи – последователно. Процесът е известен като обработка паралелно по битове, последователно по думи. Строго погледнато процеса не е паралелен, защото последователно се предават преносите от суматор j към суматор j+1.



Фиг.8-8. Обединение на суматора във верига, за паралелно сумиране по разряди.

Понастоящем това е най-популярният начин за обработка на данните (в случая сумиране) в съвременните компютри. Данните се съхраняват в паметта по начин показан на фиг. 8-9.

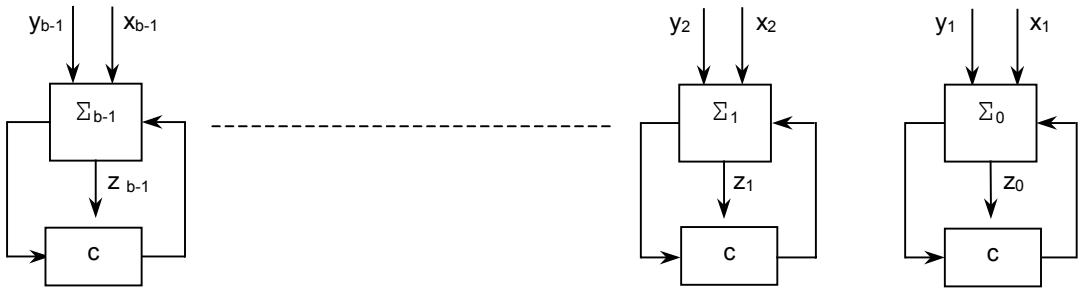


Фиг.8-9. Организация на паметта по думи.

Времето $\mathbf{T_2}$ за получаване на крайния резултат е

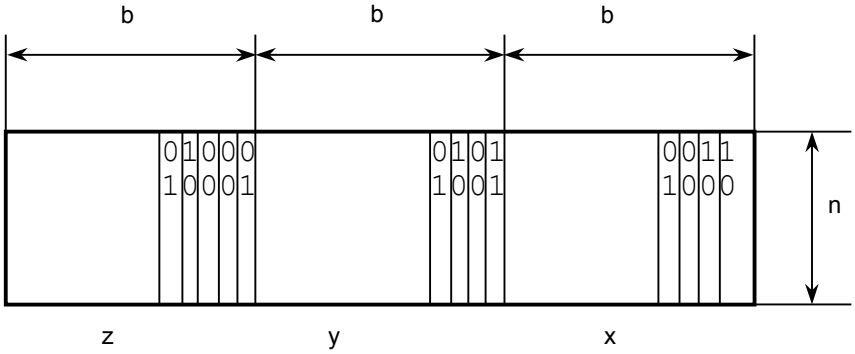
$$T_2 = n(0.5(1+b)\tau_A + \tau_M)$$

Трети начин. Устройството включва отново **b** на брой пълни суматора, но те са аранжирани по начин показан на фиг.8-10. Така всеки еднобитов суматор се използва като независим блок, който осъществява последователно сумиране на битовете на всеки елемент от **x** и **y**. Преносът от едната стъпка трябва да се съхранява в регистър, за да се въведе в следващата стъпка. Понеже са налице **b** на брой суматора, всеки от който обработва независимо от другите битовете на различни думи (в случая **b** на брой), то изчислителният процес е известен като обработка последователно по битове, паралелно по думи.



Фиг.8-10. Разрядно-секционен подход за обработка на вектори.

За да се реализира изчислителния процес по описания начин е необходимо данните да се съхраняват в паметта както е показано на фиг.8-11.



Фиг.8-11. Организация на данните за разрядно-секционна обработка.

Времето $\mathbf{T_3}$ за получаване на крайния резултат е

$$T_3 = \left\lceil \frac{n}{b} \right\rceil b(\tau_A + \tau_M),$$

където $\left\lceil \frac{n}{b} \right\rceil$ означава най-малкото цяло, по-голямо от частното $\frac{n}{b}$.

Нека сега да оценим получените резултати. Очевидно първият подход изисква най-голямо време за решаване на поставената задача. Ето защо той не намира практическо приложение, независимо от минимума апаратни средства. За да определим кой от двата останали начина е по добър, нека да намерим отношението $\mathbf{T_2/T_3}$.

$$S = \frac{T_2}{T_3} = \frac{n(0{,}5(1+b)\tau_A + \tau_M)}{\left\lceil \frac{n}{b} \right\rceil b(\tau_A + \tau_M)}$$

С цел да се опрости горния израз, да предположим, че **b** се нанася цяло число пъти в **n**. Тогава:

$$S = \frac{0.5(1+b)\tau_A + \tau_M}{\tau_A + \tau_M}$$

Полученият резултат дава възможност да се направят следните изводи:

а) Ако $\tau_A \ll \tau_M$ то двата подхода са равностойни. В началото на 50-те години, при наличието на феритна памет, е било в сила съотношението $\tau_A \ll \tau_M$ Очевидно това се явява най-вероятната

причина за това, че компютрите са се развили като разрядно-паралелни, последователни по думи.

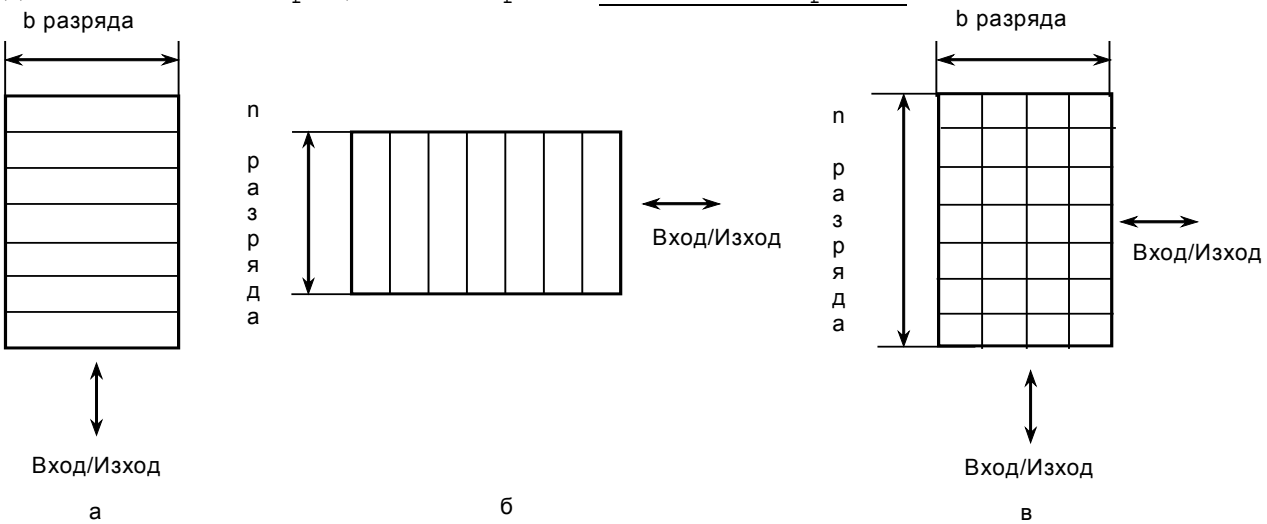
б) Ако $\tau_A \approx \tau_M$ третият подход има определени предимства. Съотношението $\tau_A \approx \tau_M$ отразява съвременното състояние, когато логика и памет се изграждат по една и съща технология и най-често се намират на един и същ чип. Ако положим **b=32** (най-често използвана ширина на думата понастоящем), то **S=8.75.** Този резултат е впечатляващ, защото производителността се увеличава почти 9 пъти само с промяна организацията на изчисление!

И така, по-високата производителност, която се получава и възможността да се включат няколко десетки хиляди паралелно работещи процесорни елементи, за да се получи масов паралелизъм, определя интереса на изследователи и проектанти на **SIMD** компютрите към еднобитовеите процесори.

3.4. Ъглово завъртане

Подробно разглеждане на паметта в паралелните системи ще бъде направено в тема 13, затова в тази точка ще се обърне внимание само на една специфична особеност, характерна за работата на процесорната матрица изградена от еднобитови процесори.

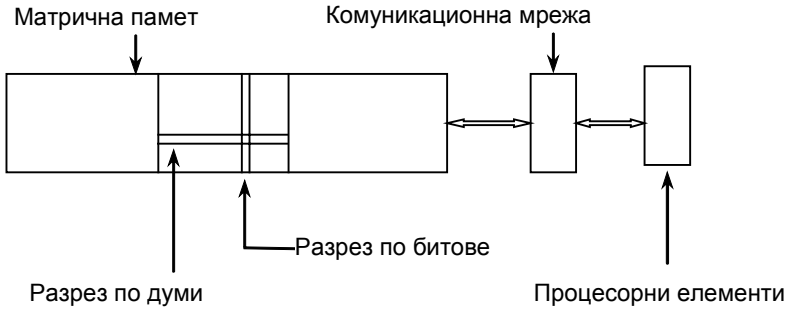
Поразрядното изпълнение на командите от процесорният елемент води към усложнен формат на представяне на данните – фиг.8-11. Тъй като понастоящем преобладават последователните компютри, данните предимно се организират с отчитане на това обстоятелство. Това означава, че данните постъпват във формат последователно по думи, паралелно по разряди, а работата на процесорният елемент изисква последователно по разряди, паралелно по думи. Следователно е необходимо да се преобразуват форматите на данните. Този процес се нарича ъглово завъртане.



Фиг.8-12. Двумерно ъглово завъртане:
а) по думи; б) по разряди; в) смесено (ортогонална памет).

На фиг.8-12а е показана класическата организация на паметта. Побитовата организация на данните с паралелни думи е показана на фиг.8-12б. На фиг.8-12в е показан пример за препокриване на тези два вида организации за съхраняване на данните.

В съответствие с изложеното по-горе, трябва да се подчертае, че ъгловото завъртване е съобразено с размерността на паметта. Функцията "ъглово завъртване" се реализира от специално устройство, включено между паметта и процесорната матрица. Естествено, това ще доведе до увеличено време за достъп до данните, а от там и до забавяне на изчисленията. В компютрите Staran и Aspro функцията "ъглово завъртане" се реализира изключително ефективно чрез използваната комуникационна мрежа, при обикновено обръщение към паметта. При този тип компютри процесорите са разположени линейно, а паметта е двумерна. Комуникационната мрежа осигурява възможност за достъп към паметта и по двата начина – по битове и по думи 8-13.

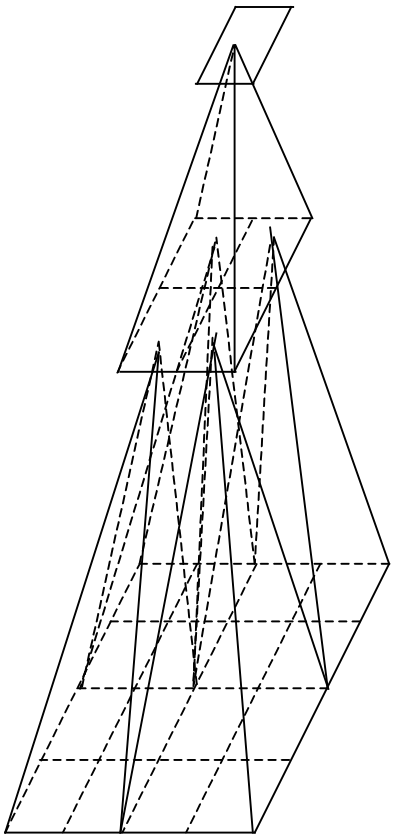


Фиг.8-13. Линейна конфигурация на процесорните елементи – двумерна памет

За голяма част от матричните системи с двумерна организация на матрицата от процесорни елементи, структурата на паметта може да се опише като тримерна. Тримерна памет е реализирана в компютрите GAPP, MPP, DAP, IUA, Blitzen. Компютърът CM-2 на Thinking Machine Corporation притежава значително по-широки възможности за настройки – шестмерна структура на процесорите, а паметта е седеммерна.

Пример. *Компютър с пирамидална архитектура.*

Концепцията за използване на пирамидалните архитектури за обработка на изображението за първи път е била предложена от L. Uhr. Пирамидалните архитектури са предназначени за решаването на специални задачи, напр. за обработка на изображението, и са пример за тясната връзка между приложението (приложенията) и архитектурата на паралелния компютър [4].



Фиг. 8-13.Пример за пирамидален процесор.

При такъв подход на най-ниското ниво компютърът представлява двумерна матрица от процесори за груба обработка на изображението. В пирамидалната архитектура, процесорите от по-високите нива са свързани със своя част от процесорите на по-ниските нива. На фиг.8-13. е показано свързването на четири процесора от по-ниско ниво с един процесор от по-високо ниво.

Колкото по-високо се предвижваме в пирамидата толкова по слабо са синхронизирани операциите. На най-високото ниво в пирамидата често се оказва компютър от тип **MIMD.**

Например, в проекта **IUA** (**I**mage **U**nderstanding **A**rray) на Масачузетския технологичен институт, на най-ниското ниво се намира паралелен асоциативен процесор **CAAPP** (**C**ontent **A**ddressed **A**rray **P**arallel **P**rocessor). Той представлява матрица от 512x512 едноразрядни процесори. Тази матрица е предназначена за изпълнение на операции над изображения на ниво пиксели. На следващото ниво също се намира асоциативен процесор **ICA**P (**I**ntermediate **C**ommunications and **A**ssociative **P**rocessor). Той е матрица с размери 64x64, 16-разредни процесори, работещи под синхронно или асинхронно управление от тип **MIMD**. На най-високото ниво на пирамидата се намира матричен процесор за символна обработка **SPA** (**S**ymbolic **P**rocessing **A**rray), представляващ 64, 32

разрядни процесора, работещи в многопроцесорен режим, който е способен да изпълнява **LISP** програми. Тази система е способна да изпълнява изчисления, представляващи въвеждане, формиране на хипотеза и верификация, анализ на неопределеност и моделна обработка.

Литература

1. Дж. Л. Портер, У.С. Милендер
Матричные супроцесорен елементр-ЭВМ.ТИИЭР, том 77, №12,декабрь 1989, стр.115-135.

2. Хокни Р., Джессхоуп К. Параллельные ЭВМ.,М.,”Радио и связь”, 1986 г., стр.145-205.

3. Howard Siegal
Analysis Techniques for **SIMD** mashine Interconnection Networks and the Effects of processor Address Masks. IEEE Transaction on Computers, Vol. C-26, No. 2, February, 1977, pp.153-161.

4. С. Гуна, Х.Уайтхауса, Т.Кайлата
Сверхбольшие интегральные схемы и современная обработка сигналов. М., “Радио и связь”, 1989, стр. 397-412.

5. Dan Moldovan
Parallel Processing: From Applications to Systems. Morgan Kaufmann Publishers, San Mateo, California, 1993

6. Geogre H. Barnis, Richard M. Brouwn, Maso Kato, David J. Kuck, Daniel L. Slotnick, Richard A. Stokes
The ILLIAC IV Computer. IEEE Transactions on Computer, Vol C17, No.8 August 1968, pp. 746-758.

7. David J. Kuck,
ILLIAC IV Software and Application Programming. IEEE Transactions on Computer, Vol C17, No.8 August 1968, pp.758- 770.

8. W. Daniel Hills, Lewis W. Tucker
The CM-5 Connection Machine: A Scalable Supercomputer. Communications of the ACM, Vol. 36, No.11, November 1993, pp.31-40

ТЕМА 9

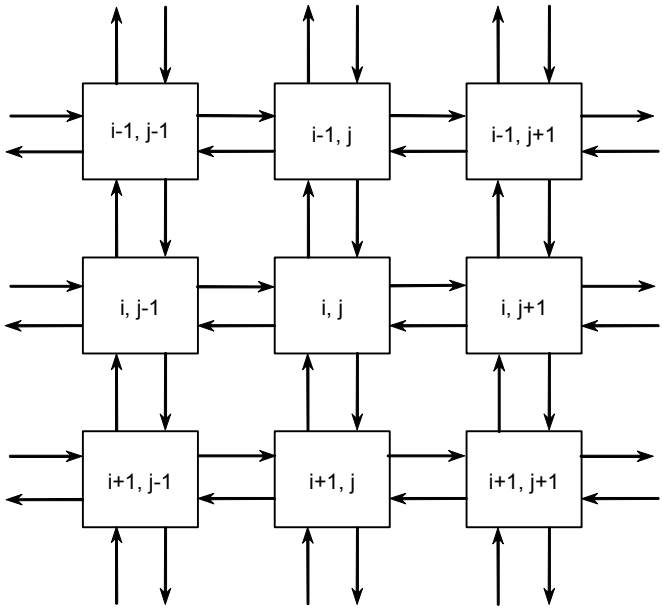
СИСТОЛИЧНИ ПРОЦЕСОРИ

1. Въведение

Чрез **VLSI** технологията е възможно да интегрират схеми със стотици хиляди компоненти върху една силициева подложка. Както беше отбелязано преди, това високо ниво на интеграция отваря пътя за масивни паралелни изчисления.

Систоличната обработка води началото от работата на Кунг и Лейсерсон от Карнеги-Мелон университета, САЩ [1]. Систоличната обработка, която по същество е конвейерна обработка в мрежа, представлява възможно решение за масивни паралелни изчисления. Нейните принципи са съвместими с **VLSI** технологията.

Въпреки, че няма строго и всеобхватно определение за систоличната матрица и систоличната клетка, следващото описание може да послужи като работна дефиниция. Под систолична матрица се разбира мрежовоподобна структура от елементи (клетки), така че да обработват данните чрез **n** размерен конвейер - фиг.9-1. За разлика от конвейера, при който само междините резултати се движат през него, тука входните и междините данни се движат през матрицата от клетки. В допълнение данните могат да се движат с различна скорост и направления. Трябва да се подчертае, че само част от клетките, тези намиращи се по границите, взаимодействат с паметта; останалите обменят данни само помежду си. Систоличната матрица обикновено има високи скорости на входно-изходните операции и е подходяща за паралелни операции.

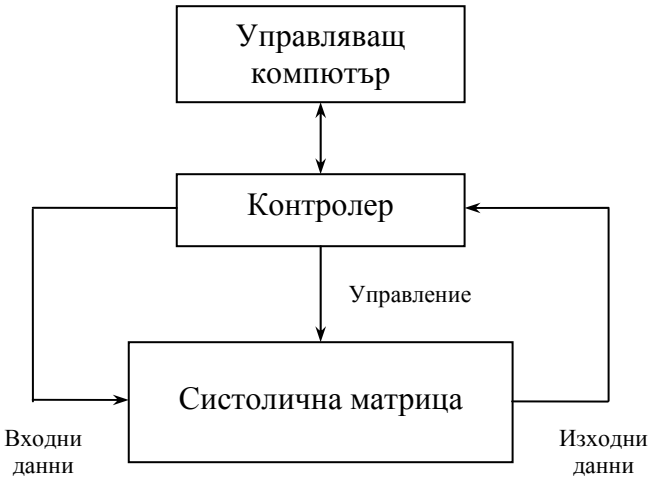


Фиг.9-1. Обобщена двумерна систолична структура.

Процесорните елементи са прости клетки, които обикновено съдържат един или два регистъра, суматор и/или умножител. За да се достигне най-добро използване на площта на силициевата подложка, клетките трябва да бъдат колкото се може повече. Също така модела на свързване между клетките трябва да бъде прост и регулярен, само с локални връзки между процесорните елементи без дълги свързвания, които биха изисквали повече площ или повече енергия за предаване на данните. Тъй като скоростта на опериране на тези матрици е обикновено много висока, всяка дълга връзка може да въведе закъснение в сигнала и затова е нежелана. Така **VLSI** структурите се характеризират чрез висока степен на модулност, отсъствие на дълги пътища от данни, локалност на връзките за предаване на данните, ограничени възможности на процесорните елементи, отсъствие на централно управление и прости синхронизиращи механизми.

Понеже систоличните матрици са високо регулярни структури, само алгоритми с повтарящи се изчисления ще се изпълняват добре върху тях. Такива алгоритми са алгоритмите използвани за обработка на сигнали, а така също и всички алгоритмите с вложени цикли. Разбира се висока производителност се достига, ако алгоритмите са подходящо настроени за систоличната матрица, което понякога изисква разработването на нови, специални алгоритми.

На фиг.9-2 е показан базовия принцип на работа на систолична система.



Фиг.9-2. Базов принцип на систолична обработка

Систоличната матрица оперира под управлението на контролера. Той осигурява управляващи сигнали и входни данни и събира резултатите. Също така той взаимодейства със главния компютър,

където се компилират програмите. Друга задача на този компютър е да поема обработката на задачи, неподходящи за систолично изпълнение. Входният поток от данни влиза в систоличната матрица, обработва се, след което напуска матрицата. Производителността на такава система силно зависи от входно/изходните характеристики. Ето защо използването на систоличните системи се препоръчва за такива алгоритми, които се характеризират с интензивни изчисления, т.е. веднъж данните извлечени от паметта е желателно да се подложат на толкова много операции колкото е възможно, преди да се запишат в паметта. I/O канал е вероятно тясното място на систоличната система.

Общото изчислително време включва действителното време за обработка и времето за предаване на данните между клетките. В систоличната матрица, поради конвейерния принцип на работа, изчислителното време трябва да е сравнимо с комуникационното време. Разпределение на изчисленията върху свързаните в мрежа процесори съставлява специфичен проблем (mapping problem). Той е по сложен дори от необходимото разделяне на алгоритъма, когато действителния размер на систоличната матрицата е по-малък от изисквания от алгоритъма.

2. Видове систолични матрици

Видовете систолични матрици се различават по:

• Топологията, т.е. начин на свързване между клетките

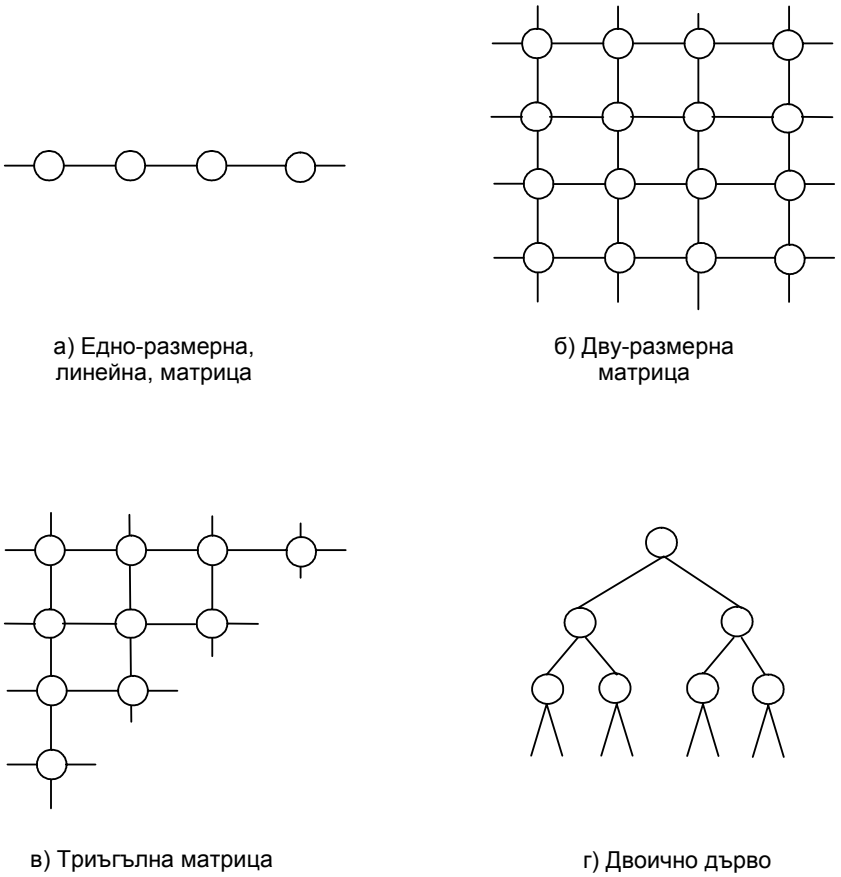
• Функционалните възможности на клетките. По този параметър те биват:

- клетки само с елементарни изчислителни възможности, т.е. клетките са прости АЛУ;
- АЛУ с няколко регистри;
- прост процесор, изпълняващ собствена програма;
- процесорен модул,в състава на който е включен процесор, памет и допълнителна логика.

• Реконфигуруеми

На фиг.9-3 са дадени някои от възможните топологии, използвани в систоличните матрици за свързвания между процесорите.

Field **P**rogrammable **G**ate **A**rrays (**FPGA**s) [14] предлага възможности за препрограмирането и реконфигуриране на матрици, които могат да бъдат конструирани за ефективни изчисление на различни проблеми. Обобщено, **FPGA** технологията е подходяща за построяване на малки систолични матрици. За специални цели АЛУ могат да бъдат конструирани и свързани в топология, най добре съответстваща на приложните цели.



Фиг.9-3. Типични систолични матрици

В таблица 9-1 е систематизирано голямото разнообразие от систолични матрици.

Таблица 9-1.

Клас	Универсални									Специализирани
Тип	Програмируеми			Реконфигурируеми			Хибридни			Хардуерни
Организация	**SIMD** или **MIMD**			**FVIMD**						**FVIMD**
Топология	П		Ф	Р		Ф	Хибридна		Ф	Фиксирана
Връзки	С	Д	Ф	С	Д	Ф	С	Д	Ф	Фиксирани
Размерност	n-размерна									n-размерна

Използвани съкращения:
П – програмируема; Р – реконфигурируема; С – статични; Д – динамични; Ф – фиксирани; **SIMD** – single instruction stream, multiple data stream; **MIMD** - multiple instruction stream, multiple data stream; **VFIMD** – very few instruction multiple data stream.

3. Специализирани систолични матрици [7]

Първоначално идеята за систолична обработка е била развита именно за матрици със специални приложения. В този случай

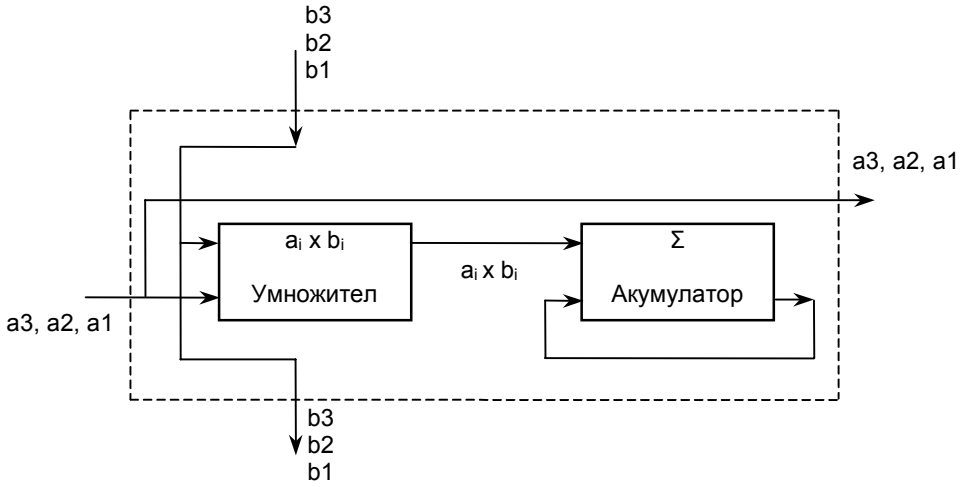
архитектурите на систоличните матрици със специално приложение се проектират за всяко приложение конкретно. Както става ясно от табл. 9-1, тук имаме хардуерно изпълнение на функциите и фиксирана топология на клетката и връзките. Трябва да се отбележи, че някои приложения не са подходящи за решаване чрез систоличния модел на изчисление, докато други, както беше отбелязано по-горе, изискват нови, специфични алгоритми за решение [6].

Една от областите, които използуват лесно систолични алгоритми са матричните операции. Например, на фигура 9-4 е дадена структурата на систолична клетка, чрез която се реализира операцията

$$s = \sum_{1}^{n} a_i b \ ,$$

където a_i ,b_i са вектори с дължина n елемента.

След като клетката е инициализирана, елементите на векторите **a** и **b** синхронно преминават през систоличния елемент, за да могат да се използват от следващите елементи. Същевременно в акумулатора се натрупва сумата от произведението $\mathbf{a_i * b_i}$. Резултатът се извежда от акумулатора в края на обработката.

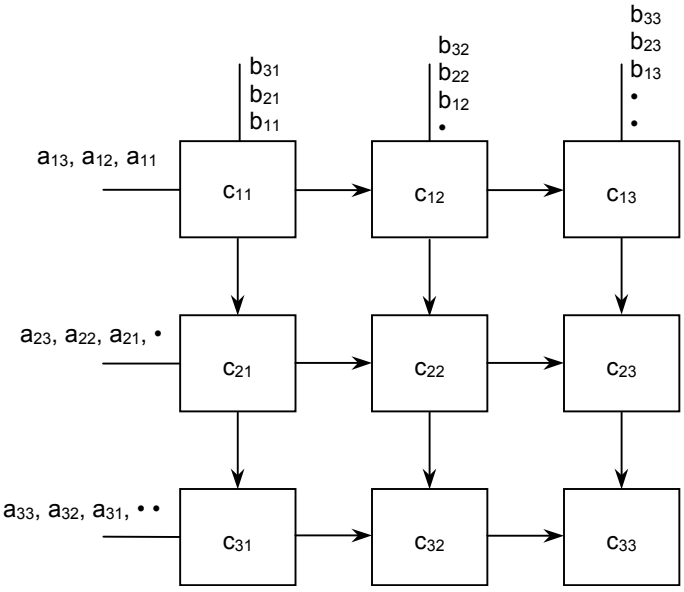


Фиг.9-4.Структура на систоличен елемент, изчисляващ скаларната сума на два вектора.

На базата на тази клетка може да се изгради систолична матрица, реализираща операцията матрично умножение. На фиг.9-5 е показано такава организация за случая на матрици с размерност 3x3. Единственото различие между едноелементната обработка и матричната обработка е, че е въведено закъснение за всеки допълнителен ред и колона на един цикъл (това е показано на фиг.9-5 с точки). В клетките $\mathbf{c_{11}}$, $\mathbf{c_{12}}$, $\mathbf{c_{13}}$ и т.н. се получават едноименните елементи на резултантната матрица. Отново както в

предходния пример, техните стойности се извеждат след завършване на обработката.

Едно от предимствата на систоличната архитектура, което се демонстрира на разгледаните примери е намалената потребност от обмен на данни между клетките и локалната памет. И действително, след като се извлече даден елемент от паметта, например $\mathbf{a_{11}}$, той се обработва не само от клетка $\mathbf{c_{11}}$, но и от $\mathbf{c_{12}}$ и $\mathbf{c_{13}}$.



Фиг.9-3. Систолично умножение на две матрици

Очевиден проблем на тази задача (и подобни на нея) е обработката на матрици с по-големи размерности от тази на систоличната матрица. В този случай матриците **a** и **b** трябва да бъдат разделени на набори от по-малки матрици, което се отразява на работата на систоличната матрица и намалява нейната производителност защото изчисленията не са чисто паралелни, а паралелно-последователни. Разбира се, развити са подходящи алгоритми за намаляване на този ефект.

Матричното умножение също така демонстрира и друг проблем, характерен за систоличните матрици със специално приложение. Действително чистото хардуерно изпълнение на функциите гарантира по-висока производителност, но цената отнесена за приложение расте, а също така намалява възможността за приложение в различни задачи. Изход от тази ситуация се търси в посока развитие на систолични архитектури с универсално приложение.

4. Универсални систолични матрици [3,11]

От табл.9-1 става ясно, че са познати три архитектурни модела – програмируем, реконфигурируем и хибриден модел – чрез които се изграждат систоличните матрици с универсално приложение.

Размерността на универсалните систолични матрици (в частност програмируемите) най-често е едноразмерна (линейна) и двуразмерна. Двуразмерната систолична матрица позволява по-ефективно изпълнение на сложните алгоритми. Поради входно-изходни ограничения, универсалните систолични матрици с размерност по-голяма от две не са разпространени.

Като пример за такива приложения могат да се посочат едни от първите компютрите **Warp** и ***i*Warp**, които са проектирани и построени съвместно от Карнеги-Мелон университета и Intel Corporation. Проектът доказва възможността на систоличната обработка.

Проектантите на **Warp** компютрите са избрали компютъра да включва няколко (само 10 клетки), но много по-сложни в структурно отношение клетки, вместо голям брой, но с по-проста структура клетки. Основната причина за това решение е било желанието да се постигне гъвкавост и способност за поддържане на широк обхват от приложения.

В състава на клетката се включва RAM памет за данни (32K думи), два регистрови файла от по 32 регистъра всеки, 3 опашки от по 512 думи всяка. Входно изходните връзки се поддържат от 6 канала.

Систоличната матрица на **Warp** има 10 клетки свързани в едноразмерна матрица, а в ***i*Warp** има 32x32 клетки, свързани като двуразмерна матрица и обща пикова производителност от 20 GFLOPS.

Литература

1. K. T. Johnson, A. R. Hurson, B. Shirazi
General-Purpose Systolic Arrays. Computer, Nov.1993, pp.20-31.
2. Мотоока Т.
Компютери на СБИС (в 2-х книгах), Мир, Москва, 1988 (1 – стр.190-194; 2 – стр. 69-74).
3. Annaratone M., Arnould E., Gross T., Kung H.T., Lam., Menzilcioglu O., Webb J.A.
The Warp computer: architecture, implementation, and performance. “IEEE Trans. Comput.”, 1987, 36, No.12,, pp.1523-1538.
4. H. T. Kung, Charles E. Leiserson
Systolic Arrays for (**VLSI**). Report April 1978: Dep. of Computer Science Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.
5. H. T. Kung, Philip L. Lehman
Systolic (**VLSI**) Arrays for Relational Database Operations. Report October 1979: Dep. of Computer Science Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.
6. H. T. Kung

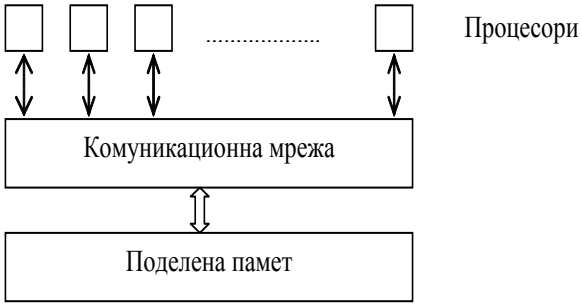
Use of **VLSI** in Algebraic Computation: Some Suggestions. Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation Snowbird, Utah, August 5-7, 1981.
7. С. Гуна, Х.Уайтхауса, Т.Кайлата
Сверхбольшие интегральные схемы и современная обработка сигналов. М., "Радио и связь", 1989.
8. Li G.J.
The design of optimal systolic arrays. "IEEE Transaction on Computers", 1985, V.34, No.1, pp.66-77.
9. Davis R., Thomas D.,
Systolic array chip matches the pace of high-speed processing. "Electron Design", 1984, V.32, No.22, pp.207-210, 212, 214, 216, 218.
10. Hyon Soo Lee, Hidekei Mori, Takakazu Kurokawa, Hideo Aiso
Systolic Array Architecture for **VLSI** FFT Processor., International Journal of Mini and Microcomputers, Vol.6, No.3, 1984, pp.49-54.
11. Dan I. Moldovan
Parallel Processing: From Applications to systems, 1993
12. H. W. Lang
Instruction Systolic Array. http://www.iti.fh-flensburg.de/lang/papers/isa
13. http://www.eleceng.ohio-state.edu/~castrop/ystolic/sosaX1-02.pdf
14. Димитър Ковачев
Идея и съображения за използване на FPGA
http://www.bgnews.bg/media.html?media=37496334

ТЕМА 10

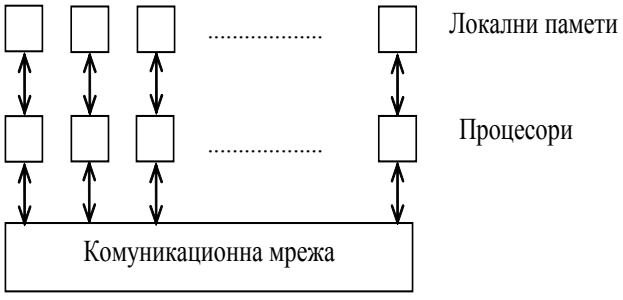
ПАРАЛЕЛНИ КОМПЮТРИ С РАЗПРЕДЕЛЕНА ПАМЕТ

1. Въведение

В тема 3, беше посочено, че една от възможните класификации на паралелните компютри се базира на степента на свързаност на процесорите. При силно свързаните системи, няколко процесора, чрез комуникационна мрежа (виж тема12) 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**ulti **P**rocessors), а вторите като компютри с разпределена памет **MMP** (**M**assively **M**ulti **P**rocessors). На фиг. 10-1 са дадени обобщените структури на двете системи.



а) Обобщена структура на SPP компютър



б) Обобщена структура на MPP компютър

Фиг.10-1. Обобщени структури на двете най-разпространени архитектури на паралелни компютри

При първият клас компютри времената за обмен на информацията между процесорите и средното време за изпълнение на машинните операции са съизмерими, докато във вторият клас времето за обмен на информацията между процесорите е по-голямо от времето за изпълнение на машинните операции [4].

SMP е един от най-зрелите модели за изграждане на многопроцесорни архитектури. Думата "симетрична" в названието на архитектурата означава, че всеки процесор може да прави всичко, което и всеки от останалите процесори. Понастоящем **SMP** често се разглежда като алтернативно название на компютрите с обща памет, която е поделена между останалите процесори и от тука идва и другата разшифровка на абревиатура **SMP**: **S**hared **M**emory **Pr**ocessors.

В **SMP** компютрите всичко, освен няколкото процесора е в един екземпляр: една памет, една операционна система, една подсистема за вход/изход. При този клас компютри, всички процесори имат пряк и бърз достъп до паметта. По такъв начин процесорът не просто работи над част от проблема, но той го "вижда" цялостно. В **SMP** архитектурата се опростява както системното така и приложното програмиране, и позволява на една многонишкова операционна система да разпределя задачите си между различните процесори и приложенията да получават толкова памет колкото е необходима. Глобалното поделяне на паметта също опростява синхронизацията на данните. Програмистите на такива системи не се грижат за това къде се намират данните, защото всеки процесор в системата може да се обръща към произволен операнд [2,3].

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 13).

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 13.

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 (от 5 до 9 включително) компютърни архитектури спадат към този клас компютри.

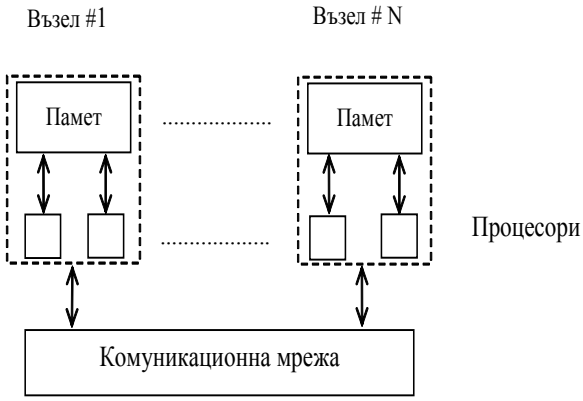
Другият клас компютри - **MMP** е с нестандартна архитектура, предвиждаща използването на разпределена памет. Тези системи

съдържат голямо количество процесори – от стотици до няколко хиляди. Всеки процесор има своя собствена памет, към която другите процесори имат непряк и по-бавен достъп. Важно е да се подчертае, че в системата няма обща памет. Така се разрешават конфликтите на процесорите с паметта. За сметка на това е 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]. Съвременни компютри, които спадат към тази архитектура са NUMA_Q на IBM, Alpha Wildfire на Compaq, MIPS64 на SGI и др.[11]. На фиг.10-2 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.10-2. Обобщена структура на компютър с NUMA архитектура

Ясно е, че всеки възел може да се разглежда като SMP компютър. По този начин, програма съхранявана в един модул на паметта, може да се изпълнява от произволен процесор в системата. Единственото различие е скоростта на изпълнение. Всички локални обръщения към паметта, направени от процесор намиращ се в същия възел, се обработват многократно по-бързо отколкото обръщенията направени към останалите модули на паметта и намиращи се в другите възли.

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.

Резонно е да се постави въпроса "Колко "неоднородна" е архитектурата **NUMA**"? Ако обръщението към паметта на другия процесор изисква време от порядъка на 5-10% повече отколкото времето към собствената памет, то една такава система от гледна точка на потребителя ще се "държи" като класическа **SMP** система и практически всички разработени програми за **SMP** компютрите ще се изпълняват и на **NUMA** компютрите. Но за съвременните **NUMA** компютри, разликата във времената за локален и отдалечен достъп е в интервала от 200-700%. При такава разлика в скоростите на достъп е необходимо да се обмисли внимателно правилното разположение на данните по различните модули памет.

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 ММР 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**).

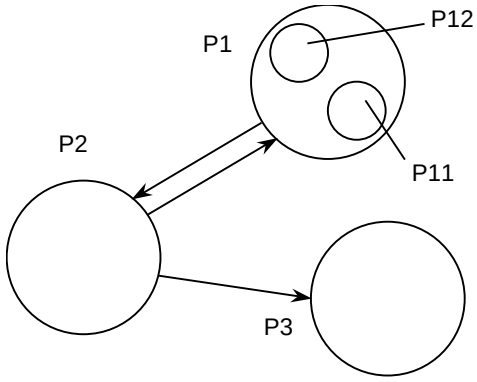
Основните елементи в **CSP** модела са процес и канал.

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.

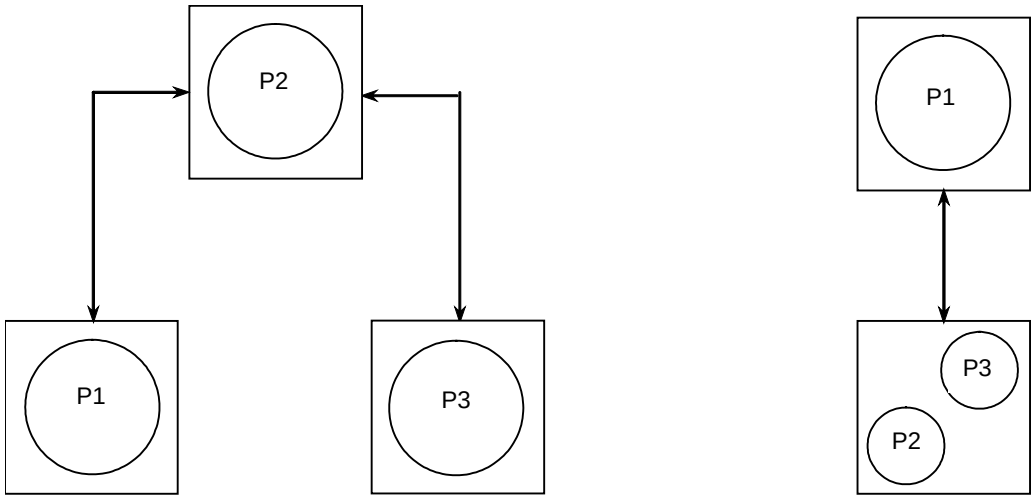
На фиг.10-3 са показани основните елементи на **CSP** модела. Процесът P1 се състои от два други процеса - P11 и P12.



Фиг.10-3. Прост **CSP** модел.

Процесите P2 и P3 комуникират еднопосочно (P2 изпраща на P3) или двупосочно P1 и P2 обменят данни и работят по кооперативен начин.

На фиг.10-4 са показани някои от възможните варианти на разположение на модела от фиг.10-3 върху различни мрежи от процесори (квадратите съответстват на процесорите, а кръговете – на процесите).



Фиг.10-4. Разположение на един и същ **CSP** модел върху различни мрежи от процесори.
а) върху три процесора; б) върху два процесора.

Моделът на Хоар по интуитивен начин съответства на паралелизма в реалния свят и е имплементиран в езика Occam.

3. Проблеми на компютрите с разпределена памет

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.

Изискванията предявявани към комуникационните мрежи се дискутират в тема 12, така че по-надолу ще бъдат обсъдени само някои от основните въпроси на програмното осигуряване.

Ако клиентите на паралелните компютри трябва да разработват машинно независимо паралелно осигуряване, то съществено се явява наличието на подходящи езици за паралелно програмиране от високо ниво и средства за тяхното поддържане. 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).

Друг проблем при разработването на подходящо приложно програмно осигуряване е така да се разбие задачата на подзадачи, изпълнявани от отделните процесори, че между тях да съществуват колкото е възможно по-малка зависимост. При системите с обща памет тази зависимост може да бъде по данни, по памет, по управление, докато в системите с разпределена памет тази зависимост може да бъде по съобщения. Много често решението на проблема се търси в разработването на подходящи алгоритми. Проблемът произтича от необходимостта, така да бъде конструиран алгоритъмът, че да осигурява възможно най-малък брой комуникации и/или комуникациите да бъдат групирани в по-големи съобщения. В потвърждение на горното 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 10-1.

Таблица 10-1

		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 авторски изследвания, проведени с процесор (транспютър) T805/30 на INMOS, а средата за програмиране е TOOLSET ANSI C.

От таблица 10-1 ясно се вижда предимството на алгоритъма с обмен на големи съобщения (груб паралелизъм) спрямо вълновия алгоритъм (фин паралелизъм). Нещо повече, последният винаги дава по-лошо време спрямо последователните изчисления, т.е в случая няма смисъл от въвеждането на паралелни изчисления. Алгоритъмът, реализиращ груб паралелизъм се оказва средно 3.5 пъти по-бърз спрямо последователния алгоритъм.

В заключение може да се каже, че този клас компютри са по-подходящи за груб паралелизъм.

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.

В отличие от другите архитектури, компютърът с разпределена памет работи като **MIMD** компютър с предаване на съобщения, при което предаването на съобщения се явява част от базовия изчислителен процес. Процесът се определя като последователни програми със системни заявки за предаване и приемане на съобщения. Писането на приложения изисква също програмистът да е наясно с организацията на паметта. Когато е необходимо, той трябва да вмъква команди за обмен на съобщения с паметта. Освен до усложняване структурата на програмата, това води до въвеждането на хардуерна зависимост в нея. По тази причина повечето производители на суперкомпютри са осигурили някаква степен на на преносимост, която се постига чрез използването на един от двата механизма за обмен на съобщения: **P**arallel **V**irtual **M**achine (**PVM**) или **M**assage **P**assing **I**nterface (**MPI**).

При този клас компютри изчисленията са разпределени по мрежата и се изпълняват едновременно във физически различни възли. Затова програмистът трябва да може да формулира задачата в термините на процеси и виртуални канали за връзка между тях. Тези съобщения изискват от диспечера на мрежата да организира ефективно предаване на съобщението от един процес

към друг. Капацитета на паметта във всеки възел ограничава броя на процесите във възела.

Операционната система, намираща се във всеки възел предоставя на процесите гъвкав набор от средства за връзка, при това механизма е един и същ както за възела така и за диспечера на мрежата. Също така се осигурява защита на процесите и предотвратява разпространението на грешки.

Определянето на маршрутите на съобщенията в мрежата са прерогативи на операционната система на възела. Поради факта, че системата е асинхронна е необходима буферизация на съобщението във възела. При получаване на заявки за приемане, съобщението се предава в буфера за съобщения на процеса.

Литература

1. H. Wiener
MIPS and reability. "Datamation", 1986, 32, №1, pp.91-
2. J. McLeod
Look out, Cray: here comes a super priced supercomputer. EUSA, 1989, №8, pp.83-86.
3. L. Curran
Choose the right parallel architecture. ED, 1989, №11, pp.41-47.
4. Н. Н. Какаулин, А. П. Пшеничников
Влияние скорости перадач информации по каналам связи на производительност распределенных вычислительные системыл М., "Финанси и статистика", 1981.
5. Р.Хокни, К.Джесссхоуп
Параллельные ЭВМ. М., "Радио и связь", 1986.
6. К. Хуан
Переспективные методы параллельной обработки и архитектура суперЭВМ. ТИИЭР, №10, октябрь 1987.
7. Gordan Bell
Ultacomputers: A Teraflop Before Its Time. Communications of the ACM, Vol.35, №8, 1992,pp.27-47.
8. Ч. Хоар
Взаимодествующие последовательные процессы. М.,Мир, 1989
9. Том Томпсън.
Най-бързите компютри в света. Byte Bulgaria 1996, бр. 1, стр.7-20.
10. В.В.Воеводин, Вл. В. Воеводин.
Параллельные вычисления. БХВ-Петербург, 2002.
11. Non-Uniform Memory Access
http://www.techgalaxy.net/Docs/Win2003/NUMA FAQs.htm
12. Behind the News
http://www.uniforum.org/news/html/publications/ufm/sept96/behindnews.html
13. Swan, Fuller Sieworek
Structure and architecture of Cm*: a modular multiprocessor., Comp., Science Research Review, CMU, 1975-1976.

ТЕМА 11

ПАРАЛЕЛНИ КОМПЮТРИ С ПОТОКОВО УПРАВЛЕНИЕ

1. Въведение

При паралелните компютри с потоково управление, машинните команди за които са готови операндите могат да се изпълнят едновременно, а възможностите за паралелно изпълнение, заложени в програмата, могат да бъдат реализирани по естествен път. Различават се два основни класа компютри:

- с управление по данни;
- с управление по заявки.

При първия клас, изчислителният процес се реализира като всяка команда чака пасивно появяването на своите операнди, за да бъде изпълнена. Програмата се състои от множество команди, които съответстват на върховете на потоковата мрежа. Изпълнението на командите в програмата се ръководи от потока данни между тях, който отговаря на движението на маркерите по дъгите на мрежата.

При втория клас изчисленията се реализират чрез заявки. При тях командата се активира, когато е необходим резултата от нея и при липса на даден операнд тя изпраща заявка към командата, която го произвежда. В този случай програмата представлява структура от вложени математически изрази. Изпълнението на програмата се състои в последователни заместявания на подизразите с техните стойности (т.нар. редукция на изрази) и изчисляване на елементарните подизрази.

Пример: *[2]. Да се разгледа изчисляването на изразите* $y_1=x_2(x_1+x_2)$ *и* $y_2=x_1(x_1+x_2)$ *при* $x_1=3$ *и* $x_2=2$

Нека с **P** означим командите в програмата, а с **V** – променливите, с която тя оперира. Плътните линии показват движението на данните, а прекъснатите – управлението при изпълнение на програмата.

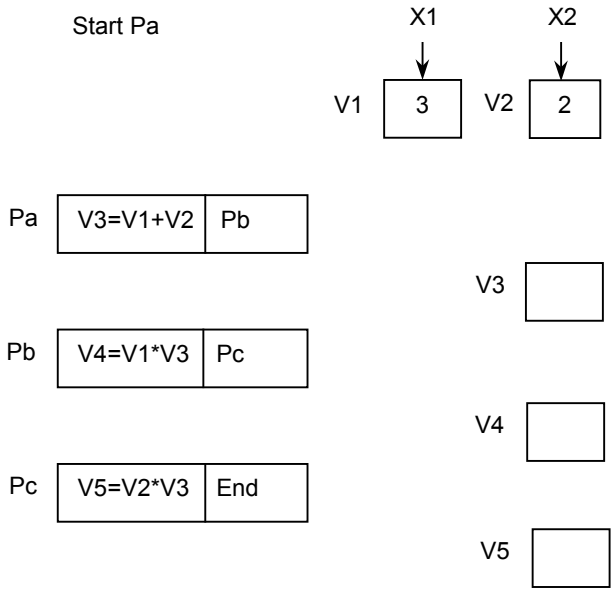
На фиг.11-1 е дадена класическата организация на изчисленията.

Изпълнение на програмата.

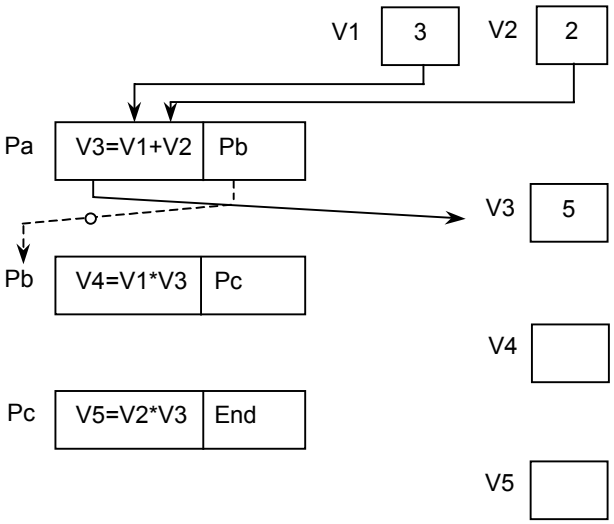
а) Командата се разрешава след получаване на управляващ маркер (празното кръгче в отделните етапи на фиг. 11-1). В апаратната реализация ролята на управляващ маркер играе специален указател – брояч на командите.

б) При изпълнението на разрешената команда се вземат стойностите от променливите, които са нейни операнди, и след това върху тях се прилага операцията на командата.

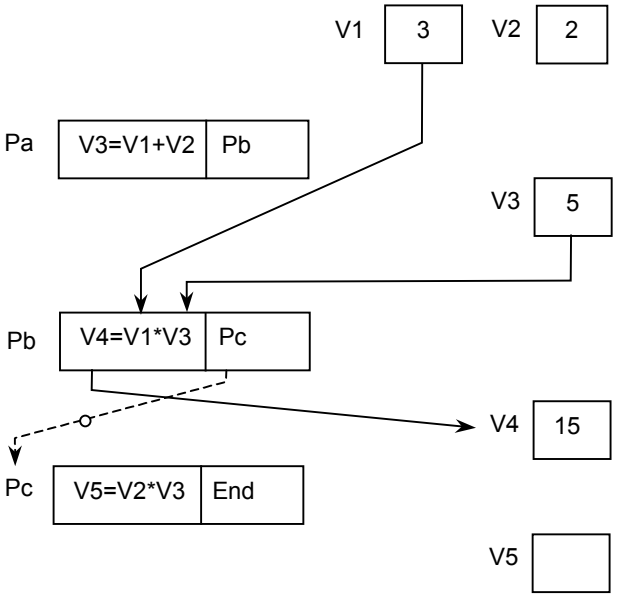
в) Полученият резултат се изпраща в променлива, предназначена за нейното съхранение.



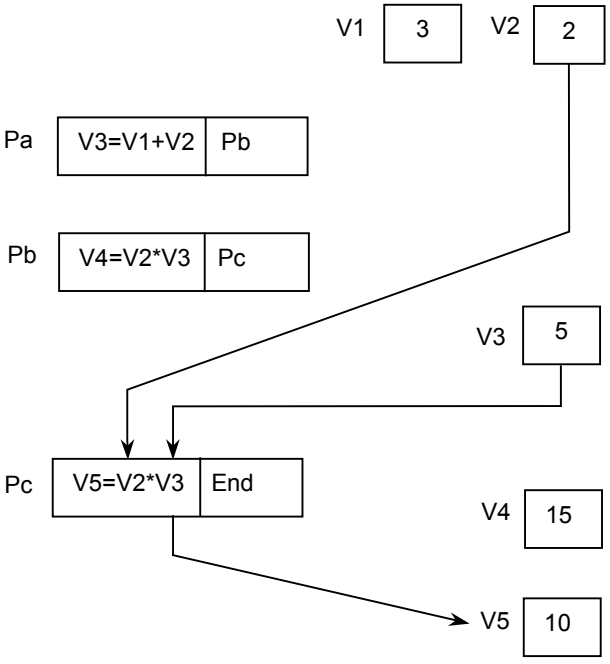
а) Начало на изпълнението.



б) След изпълнението на първата инструкция.



в) След изпълнението на втората инструкция.



г) След изпълнение на третата инструкция

Фиг.11-1 Класическа организация на изчисленията

г) След изпълнението на командата управляващият маркер се предава на предварително указана команда – като правило следващата поред в програмата.

На фиг.11-2 е показано изпълнението на програмата при потокова организация. Изчисленията се осъществяват по следния начин:

а) Всяка команда е разрешена за изпълнение, ако е получила стойностите на операндите си.

б) При изпълнението командата прилага върху стойностите на операндите принадлежащата й операция.

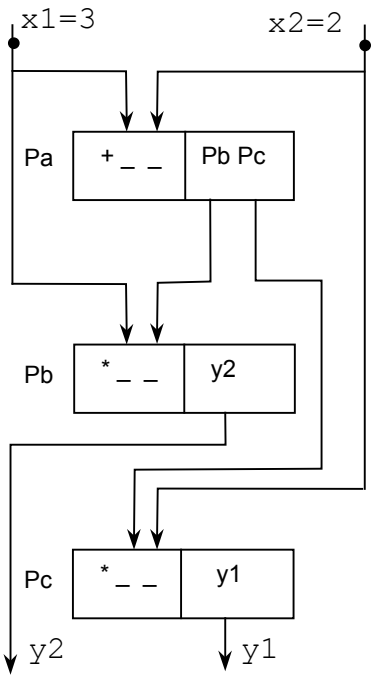
в) След изпълнението необходимото количество копия от стойността на резултата се изпраща в командите, които го ползуват като операнд – черните кръгчета на фигурата. За целта в адресната част на командата се посочват всички команди, които ползват нейния резултат като операнд. Така например, резултатът на $\mathbf{P_a}$ се ползва като втори операнд в командата $\mathbf{P_b}$ и като първи операнд в командата $\mathbf{P_c}$.

Изпълнението на програмата се осъществява, като потокът от данни преминава през различни обработващи устройства, всяко от които съответства на една команда от програмата. Стойностите на данните се предават директно между командите и изпълнението на всяка от тях се разрешава след постъпването на нейните операнди на входа на съответното обработващо устройство. По този начин управлението на изчисленията съвпада с потока данни между командите. Отделните команди могат да се изпълняват в произволен момент след тяхното разрешаване, т.е. асинхронно и независимо помежду си, като изпълнението на дадена команда не поражда характерните за класическата организация странични ефекти (не влияе на състоянието на общите клетки от паметта). То зависи единствено от стойностите на операндите и има като резултат само произведените резултатни стойности.

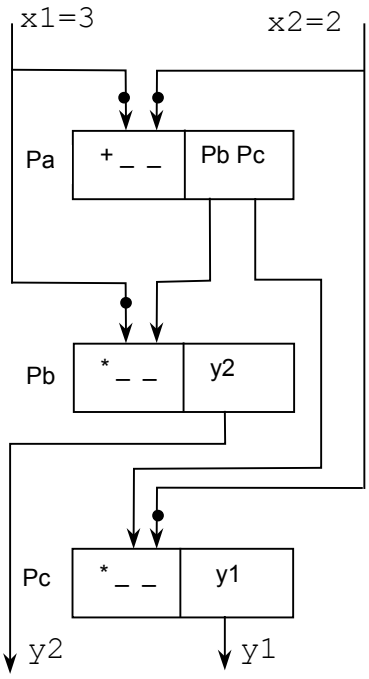
Принципът на потоковото управление води до качествено нови характеристики на изчислителната система:

- *Паралелизъм*. Потоковият принцип на изчисления дава възможност за максимално използване на наличния паралелизъм във всяка конкретна задача.

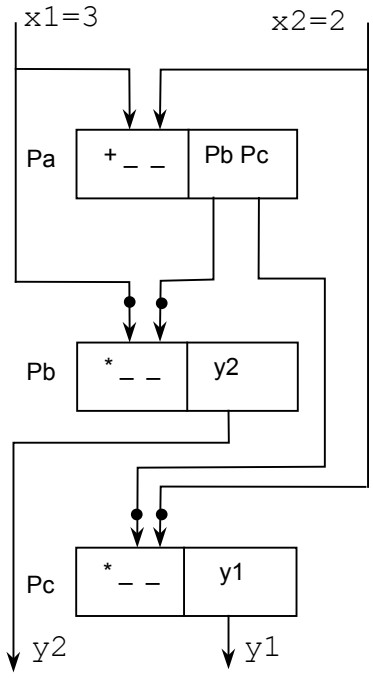
- *Модулно нарастване*. Потоковият принцип осигурява независимост на програмирането от конкретната изчислителна система. Ако например в даден момент от изчисления съществуват N разрешени команди, те могат да бъдат изпълнени от 1,2,….N процесорни елементи, при това за различно време. Следователно в компютъра могат да се добавят нови елементи (модулно нарастване), без да се променя програмирането. От друга страна, е възможно динамично, по време на изчисленията да се променя броят на модулите според нуждите на текущата програма, т.е. съществува адаптивност на системата.



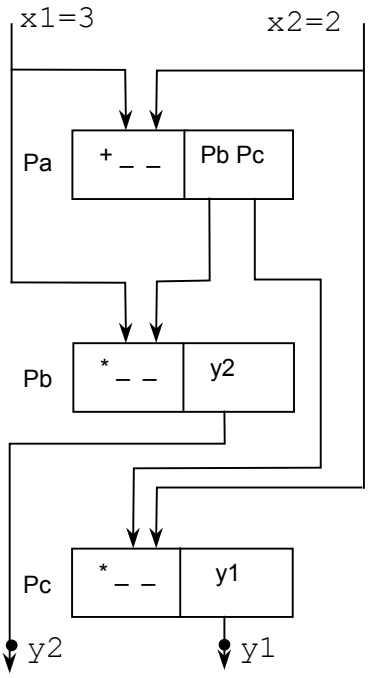
а) Начало на изпълнението



б) Pa разрешена за изпълнение



в) Pb и Pc разрешени за изпълнение



г) Край на изпълнението

Фиг.11-2 Потокова организация на изчисленията.

- *Асинхронни изчисления*. Изпълнението на даден процес (програма, процедура, функция, отделен оператор) се разрешава при наличие на входни данни за него. Разрешените процеси се изпълняват асинхронно помежду си. Изпълнението на даден процес е независимо от останалите и не може да предизвика грешки или блокировка в тях.

- *Самоидентификация на информацията.* Представянето на информацията вътре в компютъра се съпровожда с идентификатор. Той указва вида, свойствата или състоянието на информацията и се използва при нейната обработка, съхранение и предаване. Дешифрирането и обработката на идентификаторите обикновено се извършва апаратно. Самоидентификацията на различните видове информация (команди, операнди, резултати и др.) опростява управлението и прави по-гъвкава изчислителната система.

- *Разпределена обработка.* Потоковият компютър се състоят от множество изчислителни устройства (операционни елементи), които работят паралелно и асинхронно. Следователно той представлява разпределена изчислителна система. Изпълнението на отделните задачи се разпределя между компонентите на системата; така се използува паралелизма в програмата и се уплътнява работата на отделните компоненти.

2. Особености на компютрите

Моделът на изчисления, приложен в потоковите компютри, изисква нова организация на изчислителната система като цяло, а също така и на отделните нейни компоненти. Необходимостта от реализиране на асинхронен изчислителен процес с паралелни изчисления води до създаване на изчислителна система от множество независими операционни устройства и подходящи за тях памети и средства за комуникация.

2.1. Особености на управлението

Основната задача на управлението при потоковия изчислителен процес е реализирането на така наречените спускови функции. Спусковата функция се отъждествява с механизъм за разрешаване изпълнението на командите при наличието на всички необходими операнди. Според начина на реализиране на спусковите функции, потоковите компютри се разделят на два вида:

- с централизирано управление на изчисленията;
- с разпределено управление на изчисленията.

При първия вид компютри съществува централно управляващо устройство, което реализира спусковите функции, като проверява готовността на командите, избира и предава за обработка разрешените от тях.

При втория вид компютри спусковите функции на отделните команди се реализират в множество устройства (операционни или управляващи) или в клетките на паметта. Всяка команда "самостоятелно" определя своята готовност за изпълнение. Подготовка на операндите и проверката на готовността за изпълнение се извършва от устройство, отделено в момента за дадената команда, така че общото управление на изчисленията е разпределено.

Друго разделяне на потоковите компютри може да се направи на база на различните нива на реализация на спусковите функции. Известни са две нива на реализация:

- на микониво;
- на макониво.

Когато всяка команда в програмата има отделна спускова функция, се реализира управление на микониво. Така се реализират възможностите за паралелни изпълнения на отделни команди в програмата.

Когато спусковата функция се прилага на ниво части от програмата (макрокоманди, процедури, подпрограми и др.) се казва, че се реализира управление на макониво. Така се реализира паралелизъм на ниво части от програмата. Отделните части от програмата са разрешени за изпълнение при наличие на всички входни данни (аргументи) за тях.

Разбира се е възможно съчетаването и на двете нива на потоково управление в един компютър.

В зависимост от това, дали се прилага статичен или динамичен модел на изчисления, се различават статични и динамични потокови компютри. При статичен модел се работи само с едно копие на всяка команда от програмата, т.е. не е възможно повторно изпълнение на дадена команда, преди да е завършило предходното. Това отговаря на капацитет от един маркер на дъгите в статичните потокови мрежи. Възможностите за паралелни изчисления се използват толкова, колкото са зададени в машинната програма.

При динамичния модел е възможно повече от едно изпълнение на дадена команда едновременно, като се създават повече от едно нейни копия. В модела на изчисления това съответства на капацитет от повече от един маркер на дъгите, т.е. динамични потокови мрежи. Възможностите за паралелни изпълнения на копия от една и съща команда се откриват динамично в хода на изчисленията, без да са зададени в машинната програма. Така се постига по-висока паралелност на изчислителния процес в сравнение с компютрите, реализиращи статичен модел на изчисление.

2.2. Особености на изчислителната среда

Потоковият принцип на изчисление позволява пълно разкриване на потенциалните възможности за паралелни изчисления във всяка задача. Тъй като голяма част от решаваните задачи са с висока

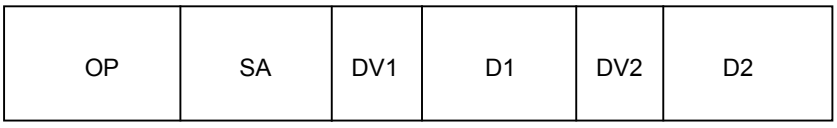
степен на паралелизъм, изчислителната среда в потоковите компютри трябва да се състои от голям брой операционни елементи (ОЕ). Всеки ОЕ представлява устройство, което може да изпълнява една команда в даден момент. Отделните ОЕ работят асинхронно и независимо помежду си, като изпълняват едновременно различни команди. При това ОЕ получава цялата необходима информация за дадена команда във вид на код на операцията, стойности на операндите и назначение на резултата, така че тя се обработва независимо от останалите команди. Изчислителната среда, образувана от множество независими ОЕ, има възможности за модулно нарастване. Това означава, че към нея може да се добавят определен брой ОЕ само чрез физическо свързване, без промяна на общата организация и на програмното осигуряване.

Според вида на ОЕ се различават три вида потокови компютри:

- Със специализирани ОЕ. В тях изчислителната среда се състои от функционално специализирани ОЕ за изпълнение предварително зададени функции.
- С универсални ОЕ. В тях изчислителната среда обединява ОЕ, всеки от които може да изпълнява произволна команда от пълния набор команда.
- С универсални процесорни модули. В тях всеки от модулите представлява ОЕ със своя локална памет. Изчислителната среда тук се интегрира със запомнящата и се изгражда от процесорни модули с универсален набор от операции.

2.3. Формат на командите

Системата от команди, основана на принципа на потоковото управление, може да се разглежда като данни снабдени с допълнителна информация. На фиг.11-3 е показана примерна структура на командата. Командната клетка съдържа значението на данните D1 и D2 необходими за операцията и по това се отличава от триадресните команди за класическите процесори, в които данните



Фиг. 11-3. Типична структура на команда за потоков компютър.
Тука означенията имат следният смисъл: OP – код на операцията; SA – адрес за резултата; D1 и D2 – стойностите на двата операнда (за двуместна операция); DV1 и DV2 – флагове за наличност на операндите.

се указват чрез адрес. Обикновено в самото начало тези данни отсъстват. Те се образуват като резултати на операции изпълнявани от други командни клетки. След като данните бъдат в наличност (записани), флаговете DV1 и DV2 се установяват в състояние

“готовност” и над данните D1 и D2 се изпълнява операцията записана в полето OP. При това резултата от операцията се записва в регистъра за данните на тази командна клетка, който е указан в SA.

2.4. Особености на запомнящата среда

Както и при другите паралелни компютри и тук запомнящата среда е предназначена да съхранява команди от програмата, входните данни, междинните и крайни резултати. Освен това обаче тя трябва да съхранява и идентификатори за всеки обект (команди от програмата, входните данни, междинните и крайни резултати). Идентификаторите представляват допълнителна информация, придадена към обекта за означаване на неговите свойства или състояние. Те съпровождат идентифицирания обект по време на изчисленията във вид на етикети на команди, имена на операнди (резултати), указатели на назначението на операнди (резултати) и др. или съществуват самостоятелно като маркери за данни или маркери за заявки.

Освен функциите по съхранение на информацията (като пасивен ресурс) запомнящата среда има активна роля в изчислителния процес. От една страна, тя участва в реализиране на спусковите функции, като проверява наличието на операндите на командите или дава необходимата информация за тази проверка чрез флагове или идентификатори на операндите. От друга страна, запомнящата среда подготвя изпълнението на командите, като комплектова цялата необходима информация за това – код на операцията, операнди и назначение на резултата във вид на стойности или чрез техните идентификатори.

От потоковия принцип на изчисления и дадените функции по съхранение и подготовка на обработваната информация произтичат следните характерни особености на запомнящата среда:

- изисква сравнително голям разход на памет за идентификатори, за копия на командите (при динамичния модел на изчисление), за копия на входните данни и междинните резултати, необходими по време на изчисленията;
- представлява активен ресурс, който изпълнява дейностите по съхранение и подготовка на информацията за обработка, а също и по управление на изчисленията (чрез спусковите функции).

Освен това съществува функционална специализация на отделните области от запомнящата среда, например памет за команди, памет за данни, памет за комплектоване на операндите, буферна памет за междинни резултати и др.

И на края, според методите за достъп до информацията, могат да се отделят три вида памети:

- с адресен достъп;
- с асоциативен достъп;
- безадресни или стекови памети.

2.5.Средства и методи за комуникация

Асинхронното действие на отделните устройства в потоковите компютри изисква силно развити средства за комуникация. Те осъществяват предаването на информация между изчислителната и запомнящата среда, между ОЕ и отделните модули памет. Предлагат се два метода за обмен – предаване на съобщения и предаване на пакети.

При първия метод между устройствата се предават блокове информация с различна дължина, всеки от които носи в себе си указател (адрес, номер) на приемното устройство. Според конкретната организация на компютъра, съобщенията представляват отделни програми, части от програми (процедури), блокове данни и др.

Предаването на пакети е частен случай на предаване на съобщения. Пакетът представлява съобщение с фиксирана дължина (определен брой байтове или машинни думи). Обикновено има няколко стандартни типа пакети в даден компютър:

- командни пакети, които съдържат една команда заедно с нейните операнди;
- операндни пакети, които съдържат стойностите на всички операнди на дадена команда;
- резултатни пакети, които съдържат резултат от една команда;
- служебни пакети, които съдържат маркери за данни или маркери за заявки.

Комуникационните ресурси в потоковите компютри най-общо могат да се разделят на три вида – устройства за управление на обмена, буфери и канали. Устройствата за управление насочват съобщенията (пакетите) според съдържащия се в тях указател към определен приемник. В различните компютри тези устройства изпълняват разпределящи, комутиращи, арбитриращи или дешифриращи функции. Буферите служат за временно съхранение на съобщенията (пакетите) при предаване между две изчислителни или запомнящи устройства. Наличието на буфери е предизвикано от асинхронното действие на тези устройства. Каналите представляват физически линии за връзка между изчислителните и запомнящите устройства на компютъра.

2.6. Особености на езиците за програмиране

Потоковите езици притежават стойностен характер, т.е. те третират данните не като обекти или променливи, а като стойности, които се поглъщат и изработват еднократно. Поради тези причини потоковите езици използват правилото на дефиниционните езици за “еднократно присвояване”.

Стойностният характер на потоковите езици определя едно от най-важните им свойства – липсата на странични ефекти при тяхното използване. Това ще рече, че изпълнението на произволен оператор

в хода на изчисленията не може да повлияе отрицателно върху изпълнението на кой да е друг оператор (при конвенционалните езици използването на дадена променлива преди стойността ѝ да е установена - било то чрез оператор за присвояване или чрез оператор за четене (въвеждане), води до съществена грешка). Тези езици, за разлика от функционалните, позволяват именуване на стойности, което винаги е с локален характер и по този начин се облекчава писането и четенето на програми.

Потоковите езици подобно на конвенционалните оперират с множество от данни - елементарни и структурни. Отношението към структурните данни е еднакво с това към елементарните. Структурните данни също се разглеждат като стойности и промяната на произволен елемент от една структура винаги води до създаването на нова стойност от тази структура.

Потоковите езици могат да бъдат ориентирани както към конкретна архитектура (специализирани езици), а така също и към определен клас от тези архитектури (универсални езици). Примери за потокови езици са LAU (Тулуза, Франция), VAL (Масачузетския технологичен институт, САЩ), Id (разработка на университета в Ървин, Великобритания). Първите два са предназначени за компютри от статичен тип, а третия – от динамичен тип.

3. Структура на потоковия компютър

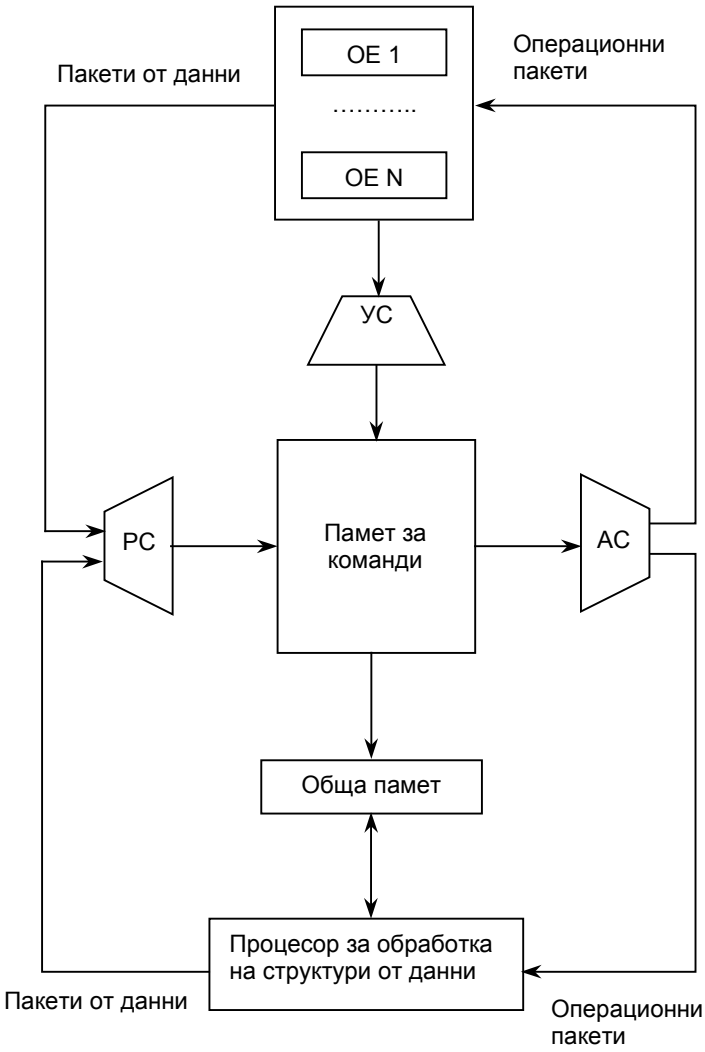
Съществуват три способа за организация на структурата на компютъра управляван от потока данни:

- Най-прекият подход се заключава в взаимното съединение на множеството ОЕ, така че да се реализира изчислителния процес. Програмата при това се задава посредством комутация между ОЕ (подобно на това както се прави в аналоговите машини; така се програмирали компютрите до появяването на принципа за съхраняване на програма в паметта на компютъра). Тука е възможно да се осигури абсолютната максимална скорост на обработка, но програмирането е подобно на електрическо съединение на операционни блокове и затова е лишено от универсалност.

- Конфигурацията от връзки, необходими на ОЕ се формира чрез матричен превключвател (виж тема 12), управляван от програмата. Този способ на организация притежава универсалност. Но при увеличаване на броя операции в програмата, управлявани от данните, сложността на връзките между ОЕ стремително расте, което възпрепятства реализацията на дадения подход с апаратни средства.

- В отличие от изложените по-горе способи, които реализират връзките по физически път, може да се използва универсален подход, при който съединенията между ОЕ се реализира по логически път с помощта на комуникационна мрежа. На практика това е единственият способ за структурна организация на компютъра.

Без да се включват входно-изходните устройства, в потоковите компютри съществуват четири основни структури – с обща шина, разпределена, кръгова и йерархична.



Фиг.11-4.Обобщена кръгова структура на паралелен компютър с потоково управление.

Отделните устройства (изчислителни, запомнящи, управляващи) в компютрите с обща шина се свързват посредством канал (шина), който се използва последователно, в режим на времеделение, между устройствата.

За компютрите с разпределена структура е характерно наличието на множество еднакви модули, свързани по различни

начини помежду си. Модулите са универсални ОЕ със своя локална памет.

Кръговата структура работи като цикличен конвейер от изчислителни, комуникационни и запомнящи устройства. Между тях обикновено съществуват множество паралелни еднопосочни канали. Характерно за такава структура е, че изчислителните устройства се състоят от независими ОЕ, които работят паралелно. Запомнящите устройства са специализирани (за команди, за операнди и др.) или обединяват модули с паралелен достъп. В някои случаи в структурата е включено и управляващо устройство – фиг.11-4.

Основните структури се срещат и в комбинирана форма на различни нива в един компютър. Така например, разпределена структура се прилага заедно с кръгова вътрешна структура на ОЕ.

4. Проблеми на компютрите с потоково управление

Тук ще бъдат дискутирани някои от най-основните проблеми на компютрите с потоково управление. Тези проблеми са следствие на принципите заложени в работата на този тип компютри.

Първата група от проблеми е свързана с обръщенията към паметта. Бекус отбелязва като недостатък на традиционните компютрите, наличието на многочислени потоци информация между процесорите и основната памет, нямащи непосредствено отношение към обработката на данни. Този проблем не е решен и при компютрите с потоково управление. За конвенционалните компютри е характерна висока локалност на обръщенията към паметта, позволяваща ефективно използване на кеш-паметта и структурирането и на няколко нива (йерархична памет). Освен това в тях е възможна реализация на изпреварващо управление на основата на конвейер за командите и широко използване на регистри. Реализацията на тези прийоми в компютрите с потоково управление е свързано с принципни трудности. В частност, съвместна работа на множество ОЕ изисква изключително висока пропускателна способност към паметта. За отстраняване на тесните места при работа с паметта се предлага повишаване на функционалното ниво на ОЕ, но определянето на това ниво само по себе си е проблем.

Друга група от проблеми при компютрите с потоково управление се проявява при обработката на регулярни структури. Тука масивът се разглежда като набор от елементи, обработвани паралелно, т.е. има понижаване на функционалното ниво на векторните (матричните) операции до скаларни. Като резултат от това обръщенията към паметта са с произволен (рандомизиран) характер и не може да се използва ефекта, постигнат за сметка на висикоскоростното многоканално обръщение към линейно организираната памет постигнат чрез техниката на разслоението (виж тема 13). Освен това, поради необходимостта от обработка на индекси и други действия в компютъра се появява голямо количество допълнителни разходи. От казаното следва, че е целесъобразно включването на функционален

блок от високо ниво, ориентиран към обработка на регулярни структури.

И на края третият кръг от проблеми е свързан със системата от прекъсвания. Както е известно, неразделна част от класическия компютър е неговата система за прекъсване. Поради липса на такъв апарат в потоковите компютри, за да се реагира на изключителни ситуации се използва друг механизъм. Множеството от данни в потоковите езици се допълва със специални стойности-грешки. Ако при аритметична операция се получи препълване, се изработва "стойност-грешка-препълване", която се изпраща на оператора приемник. Това е друг обработващ оператор или управляващ оператор, който ще направи разклонение в изчислителния процес в съответствие с настъпилата грешка.

Литература

1. Т. Мотоока
Компьютеры на СБИС, 2-х книгах, М., Мир, 1988, 1т., стр.89-110
2. К.Боянов и др.
Изчислителни системи с паралелна обработка на данните., С., Техника, 1986.
3. К. Боянов и др.
Изчислителни системи с потоково управление., Автоматика и изчислителна техника, No4, 1983, стр.25-35.
4. Амамия М., Танака Ю. Архитектура ЭВМ искустенный интеллект., М., "Мир", 1993 г., стр.176-222.

ТЕМА 12

КОМУНИКАЦИОННИ МРЕЖИ

1. Въведение

Една от особеностите на паралелните компютри е необходимостта от развита среда на обмен на информацията между процесорите за да се получи крайното решение на задачата. Прието е тази среда да се нарича обобщено комуникационна мрежа (КМ) Недостатъчното бързодействие и/или ограничените комбинационни възможности на КМ могат съществено да снижат очаквания ефект от използването на паралелния компютър. 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.

От гледна точка на архитектурата на паралелния компютър именно последния подход представлява най-голям интерес, ето защо по-надолу ще се разглежда само този подход. Ясно е обаче, че на практика трябва да се намери общ баланс между посочените по-горе подходи, за да се изгради КМ с минимални комуникационни загуби.

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.

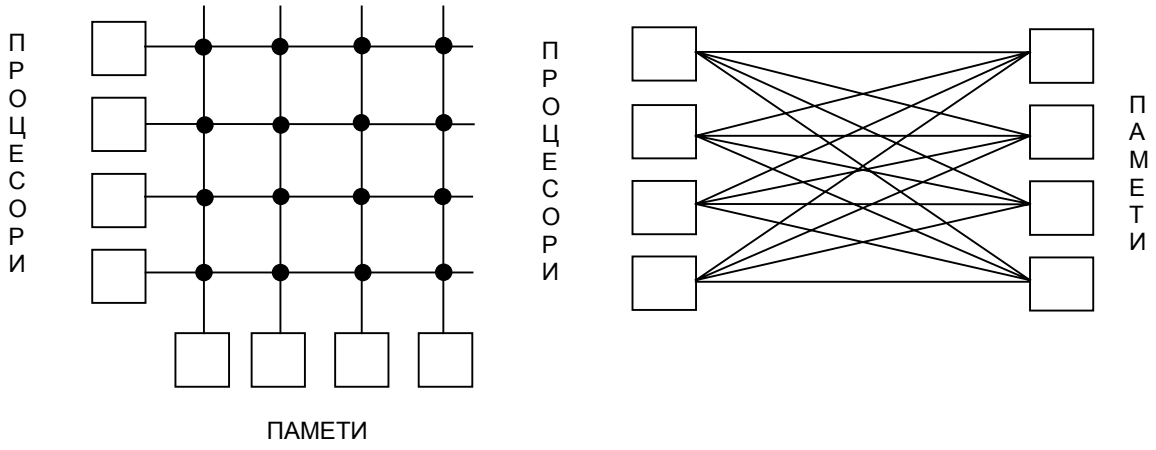
КМ могат да осигуряват комутация:

- 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.12-1. 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.12-1. Типични представители на КМ по пространство:

Стремежът да се съчетаят положителните страни на двата класа КМ води до трети клас КМ с комутация по пространство и по време.

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.

Режим на работа. Известни са два режима на работа - синхронен и асинхронен. Синхронна комуникация е необходима за процеси, в които комутационните връзки са установени синхронно за транслационно предаване на данни (команди) или предаване на функции за обработка върху данните (напр. **SIMD** компютрите). Асинхронна комуникация е необходима при множество процеси, за които връзките се създават динамично. КМ може да бъде

проектирана така, че да улеснява и двата режима. Следователно има три режима на работа – синхронен, асинхронен и комбиниран.

Стратегия на управление. Чрез вътрешносистемни комуникационни функции, специфични за всяка КМ, се определя пътят за свързване между i-тия вход и j-тия изход на мрежата. Тези комуникационни функции влияят върху състоянието на КМ. В зависимост от това къде се изчисляват тези комуникационни функции, управлението може да бъде централизирано и децентрализирано. В първият случай има специално устройство, което управлява превключванията в мрежата. Във вторият случай такова устройство няма, а самите КЕ извършват необходимите изчисления. Последната технология се нарича още разпределено управление.

Превключваща методология. Има две методологии – схемно превключване (канал) и програмно превключване (пакетно превключване). При схемното превключване съществува физически път между източника и приемника, който се поддържа през цялото време на сеанса на предаване. При програмното превключване данните се поставят в пакет и се предвижват през КМ без да се установява физически път. Обобщено, схемното превключване не е много подходящо за предаване на голям обем от данни, а програмното превключване - за къси съобщения.

Мрежова топология. КМ може да бъде описана като граф, чийто върхове съответстват на КЕ, а дъгите - на ЛВ. Мрежовата топология се явява ключов фактор в определянето на възможностите на КМ от гледна точка на архитектурата, затова на нея ще се спрем по-подробно.

3. Топология на КМ

Топологията на КМ бива; регулярна и нерегулярна. В паралелните компютри намира приложение само регулярната топология, ето защо по-надолу се разглежда само тя. От своя страна, топологиите, които са регулярни биват статични и динамични.

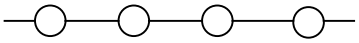
3.1. Статични комуникационни мрежи

Статичните комуникационни мрежи (СКМ) осигуряват непосредствена връзка само между строго определени компоненти на мрежата. Връзки с останалите компоненти се осигуряват индиректно. Като правило СКМ работят с комуникация на пакети, при което процесорите играят ролята на КЕ. СКМ биват едномерни, двумерни, тримерни,… многомерни. На фиг.12-2 са дадени някои примери на най-често срещаните СКМ.

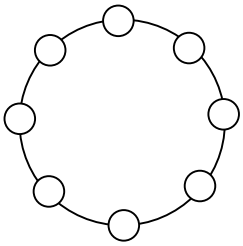
За сравнителна оценка на различните топологии СКМ се използват следните параметри:

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

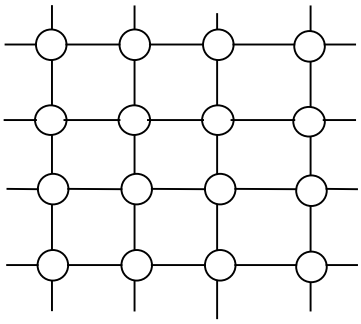
Диаметър на мрежата – D. Дефинира се като максималното разстояние между произволна двойка върхове в пътя свързващ двата върха. В случая под разстояние се разбира минималния брой на дъгите, свързващ двата върха. Например, за СКМ тип



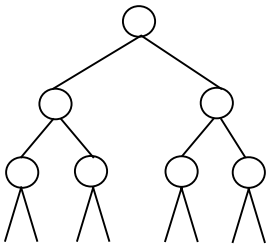
а) линийка (конвейер)



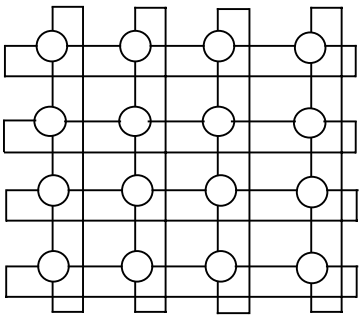
б) кръг



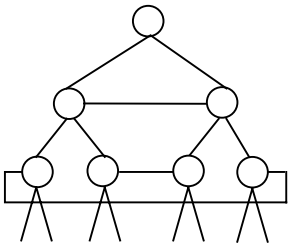
в) решетка



г) двоично дърво



д) тор



е) хипердърво от тип I

Фиг.12-2. Примери на статични комуникационни мрежи.

"кръг" от фиг.12-2б, D=4, а за СКМ тип "решетка" – фиг.14-2в, 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, т.е.

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

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 12-1, но за по-голямата част от тях определянето им се реализира чрез използуването на известен алгоритъм за намиране на пътищата в граф, напр. на Форд или Дейкстра и др.

Таблица 12-1.

Тип КМ	D
Линийка	N
Кръг	N/2
Квадратна мрежа	$2(\sqrt{N}-1)$
ILLIAC IV	$\sqrt{N}-1$
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}.

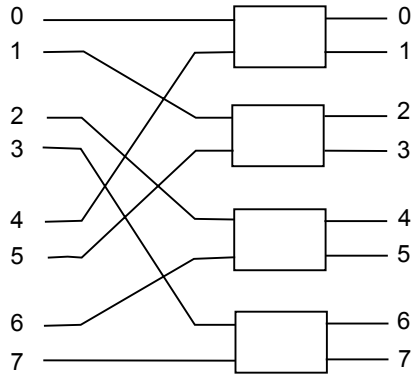
3.2. Динамични комуникационни мрежи

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}→{N}, което се тълкува като размяна F на множеството разместванията {N} от N елемента. ДКМ биват [3]:

- едностъпални
- многостъпални.
- матричен превключвател

Едностъпалните мрежи - фиг.12-3, се наричат още рециркулационни, тъй като при реализация на информационния обмен може да възникне необходимост съобщенията да циркулират многократно през стъпалото до достигане на местоназначението.



Фиг.12-2. Едностъпална КМ 8x8 от тип shuffle-exchange

Като правило, едностъпалните ДКМ намират приложение при изграждането на по-сложните, многостъпални, мрежи.

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

- едностранни
- двустранни.

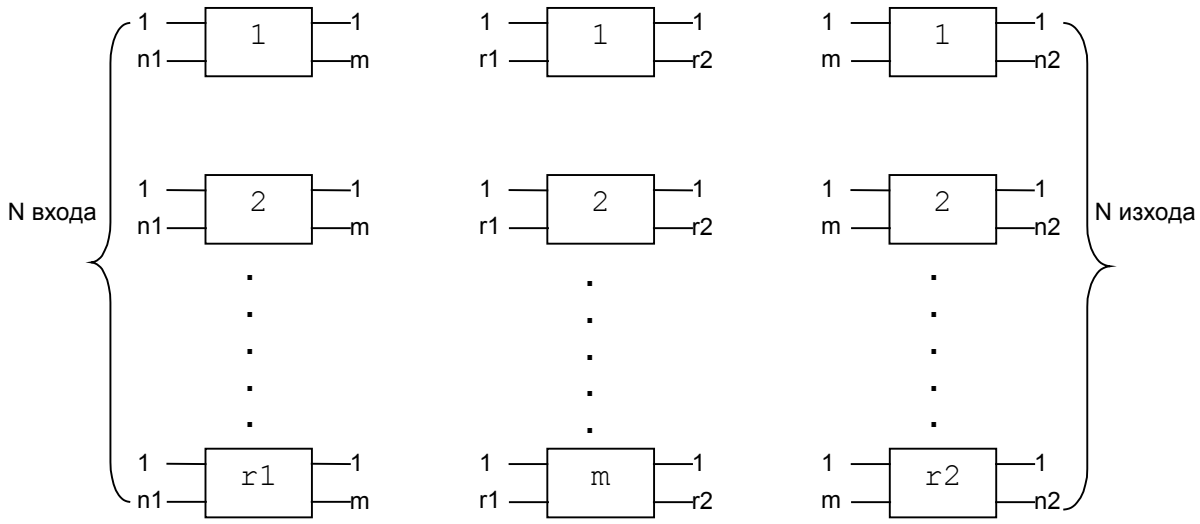
Едностранните мрежи използуват двупосочни ЛВ, а двустранните имат входна страна и изходна страна. Последните от своя страна се разделят на блокиращи, неблокиращи с реконфигурация и напълно неблокиращи.

В блокиращите ДКМ е възможно с присъединяването на дадена една двойка предаватeл-приемник да предизвика конфликт в използуването на мрежата за друга двойка предаватeл-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 предаватeл-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 общият случай. Като следствие на това се получава:

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

Модел на двустранна ДКМ при k=3 стъпала е показан на фиг.12-4.



Фиг.12-4. Обобщен модел на многостъпална КМ.

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 $r_2xn_2=N$).

Връзките между КЕ в различните стъпала се определят от вътрешно системните функции за връзка, които в крайна сметка дефинират и различните видове ДКМ (мрежа на Клос, мрежа на Бенес, омега мрежа, делта мрежа, сигма мрежа, Мемфис мрежа и т.н.).

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.

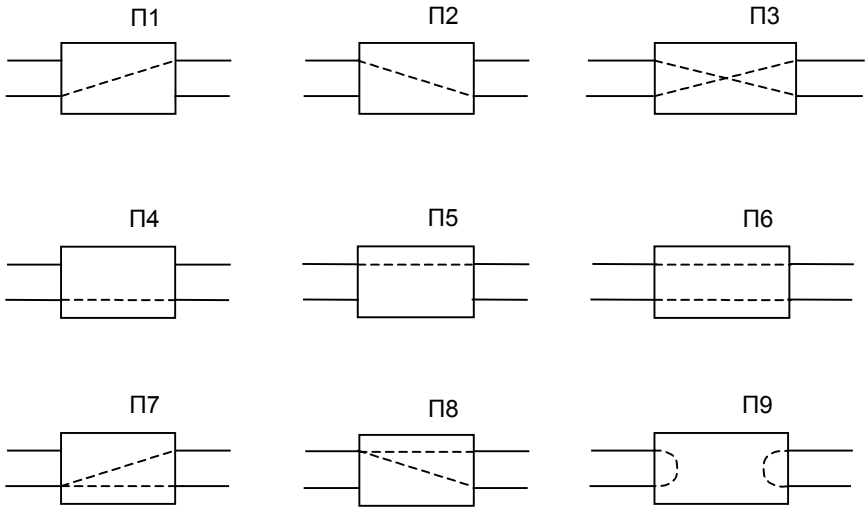
Сегментиране. Това е разделяне на ДКМ на независими подмрежи с различни размери. Всяка подмрежа трябва да има всички възможности за връзки на цялата мрежа от същия тип и размер.

Много автори подчертават важността на този параметър. Различни изследвания показват, че едностъпалната мрежа като shuffle-exchange или мрежата ILLIAC IV не могат да се сегментират, докато data manipulator може.

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.

4. Архитектура на комуникационния елемент

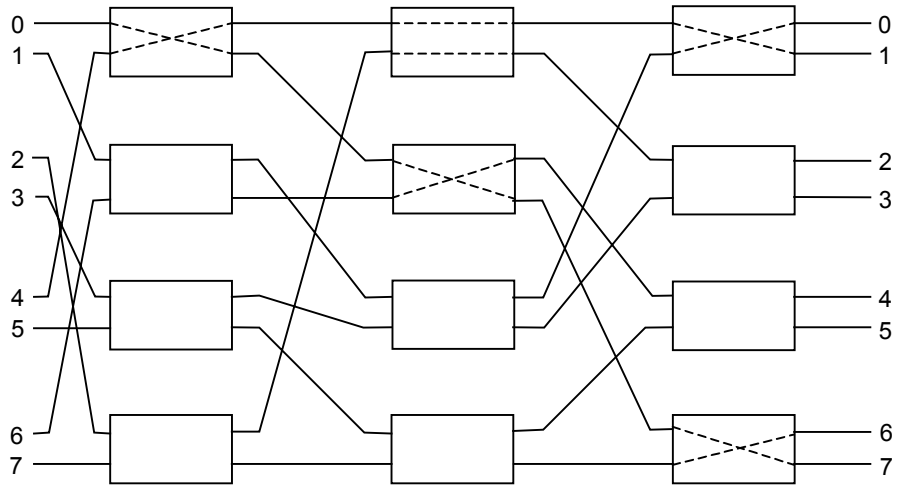
По сложност на апаратната и програмната си реализация КЕ се доближава до сложността на стандартен микропроцесор. Вътрешната структура на КЕ включва буферна памет, адресни регистри, цифрови компаратори, мултиплексори, комутаторен елемент, локален контролер и приоритетна логика. Размерът на буфера определя в голяма степен параметрите на мрежата (пропускателна способност и закъснение), като обуславя непосредствено степента на блокиране в мрежата поради възникване на конфликти за ползване на общи комуникационни ресурси. Препоръчва се регулирането на трафика в мрежата да се реализира не чрез увеличаване на буферните ресурси, а посредством приоритетна дисциплина на обслужване на заявките. Най-широко разпространение са получили КЕ елементи с два входа и два изхода. Възможните им състояния са показани на фиг.12-5.



Фиг.12-5. Възможни състояния на КЕ от тип 2x2.

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



Фиг.12-6. Delta мрежа за N=8.

Всеки КЕ може да заема едно от двете състояния $П_3$ или $П_6$ от фиг.12-5, в зависимост от съвпадението ($П_6$) или несъвпадението ($П_3$) на i-тия бит на адресите на предавателя-приемника. Например, нека да се предположи, че трябва да се изпратят данни от източник с адрес 0 (двоичен код 000) към приемник с адрес 7 (двоичен код 111). Тогава управляващата функция (тяг) е T=111 и състоянията на съответните КЕ е дадено на фиг.12-6. Да предположим, че паралелно с това предаване е необходимо да се осъществи предаване между двойката терминали $4 \rightarrow 1$ (в двоичен код $100 \rightarrow 001$). Тягът в този случай е T=101 и отново на фиг.14-6 са посочени състоянията на съответните КЕ. Вижда се, че двата потока данни минават през един и същ КЕ (най-горният от първото стъпало), но поради еднаквостта на необходимите превключвания (размяна), не настъпва конфликтна ситуация. Но ако вместо предаване от $4 \rightarrow 1$ е необходимо да се предава $4 \rightarrow 5$, което определя тяг T=001, то същият този КЕ трябва да бъде в другото състояние (предаване направо), което се явява конфликтна ситуация и води до блокиране на мрежата.

За решаването на този проблем в случая трябва второто предаване да се отложи във времето, докато завърши предаването 0→7.

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≤i≤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

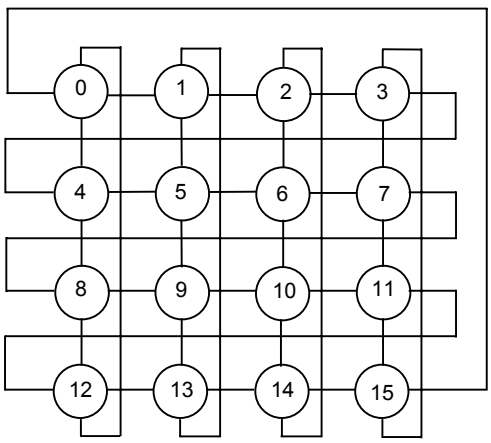
Тази мрежа се използва в паралелния компютър ILLIAC IV и се описва от четирите функции:

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

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

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

На фиг.12-7 е показана такава мрежа за N=16.



Фиг.12-7. Комуникационна мрежа ILLIAC IV с размер 4x4

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: $cube(q_{d-1}...q_{i+1}q_iq_{i-1}...q_0) = q_{d-1}...q_{i+1}\overline{q}_iq_{i-1}...q_0$ където $Q=q_{d-1}....q_{i+1}q_iq_{i-1}....q_0$, $0\le Q<N$, $0\le i<d$, а $\overline{q}_i$ 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:

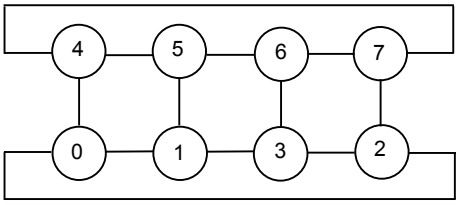
а) Програмата лесно може да бъде построена така, че тя да се изпълнява без изменение на хиперкуб с произволен размер.

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.

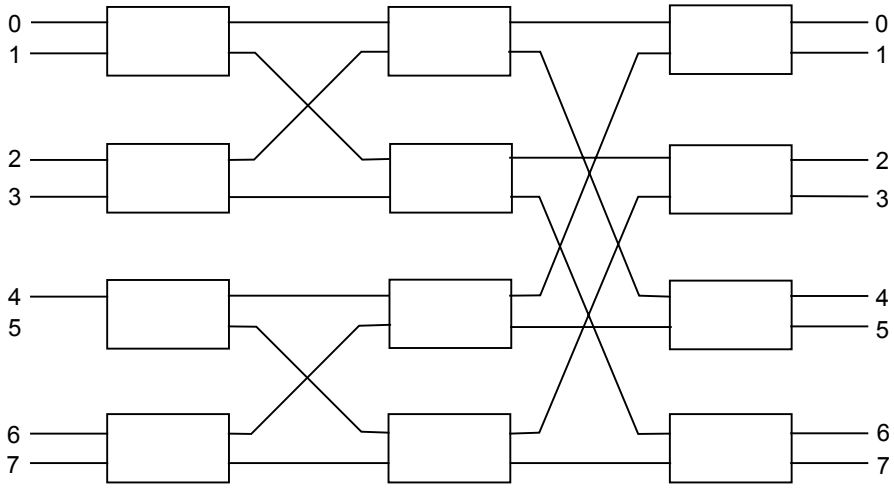
б) Хиперкубът с по-голяма размерност може да бъде лесно използван от много клиенти едновременно, на всеки от които операционната система назначава подкуб.

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.

Двоичната хиперкубична мрежа може да бъде приложена както като статична КМ – фиг. 12-8а, така и като многостъпална мрежа – фиг.12-8б.



а) едностъпална мрежа от тип двоичен куб



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

Фиг.12-8. Примери за двоичен куб.

Обобщено, многостъпалната мрежа от тип двоичен куб се състои от d стъпала всяко съдържащо N/2 КЕ. Всеки КЕ може да заема едно от двете състояния $П_3$ или $П_6$ от фиг.12-5, в зависимост от съвпадението ($П_6$) или несъвпадението ($П_3$) на i-тия бит на адресите на предавателя-приемника.

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 $d=\log_2 N$.

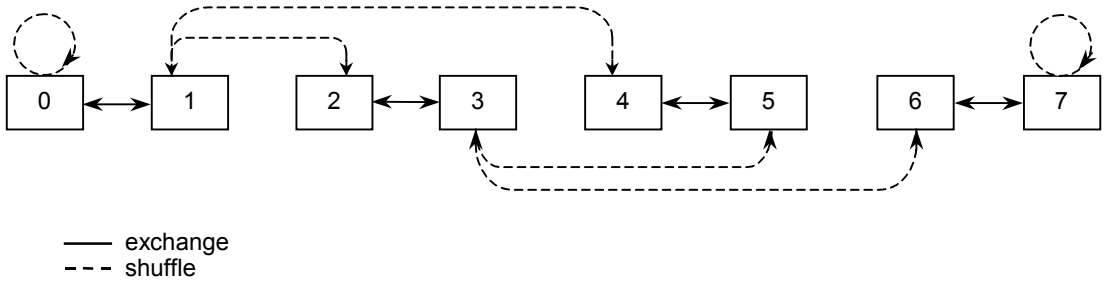
Например, за $N \ge 4$, shuffle(1)=2.

Функцията **exchange** се дефинира като:

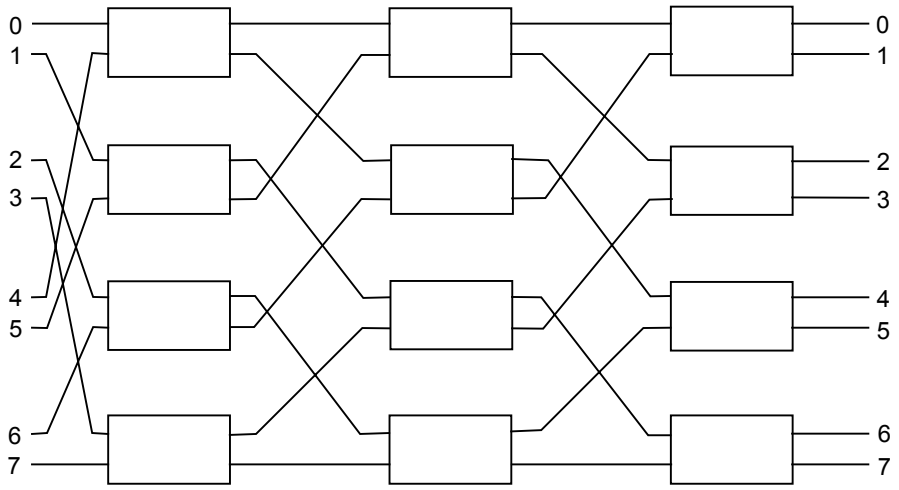
$$\textbf{exchange}(Q)=\text{cube}_0(Q),$$

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.

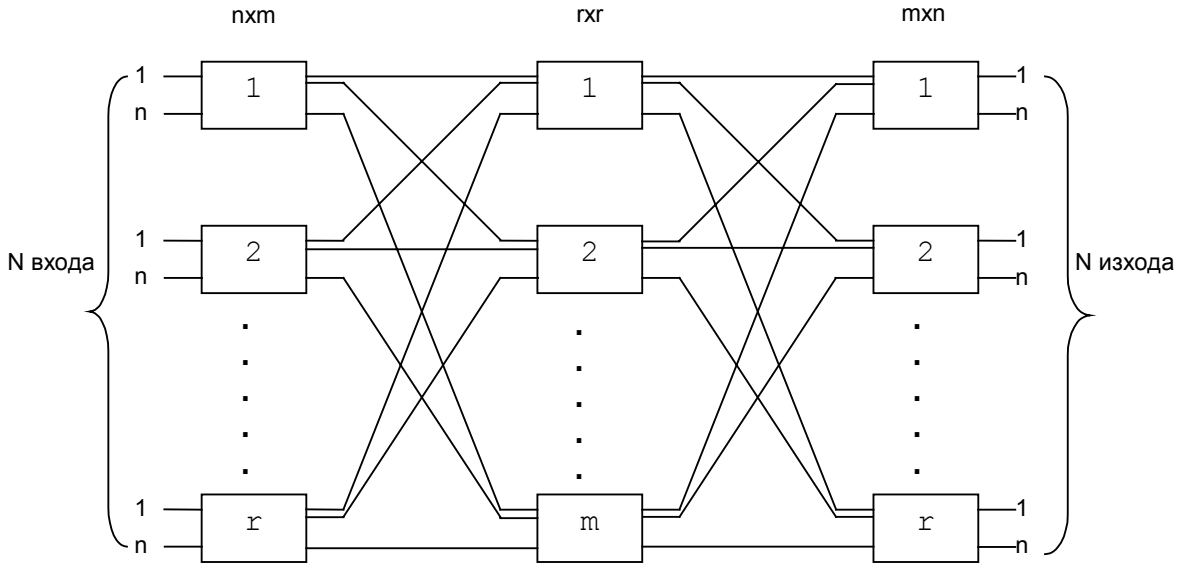
Фиг.12-9. Примери за мрежи от тип shuffle-exchange.

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≤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.

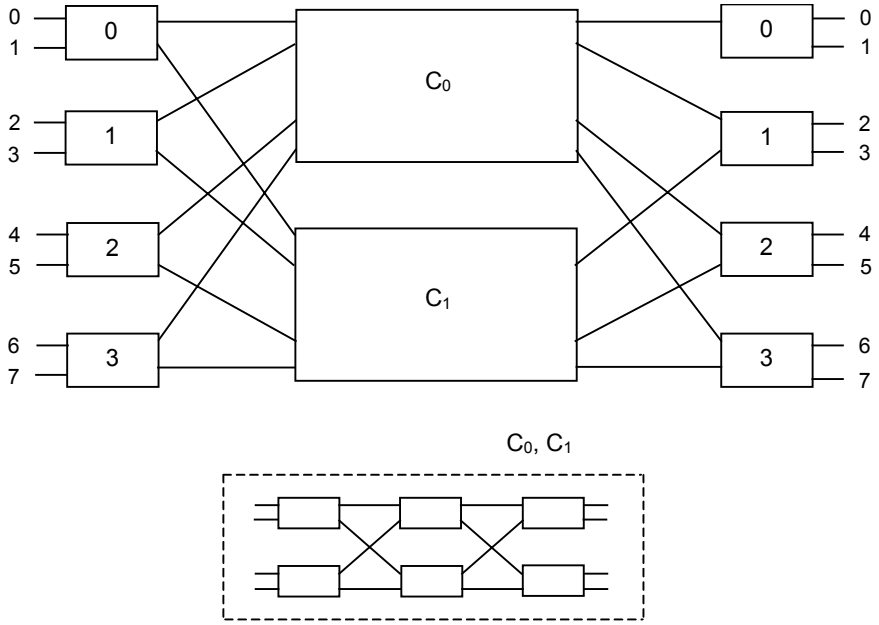
Мрежата на Клос има структура показана на фиг.12.10. КЕ тука са превключватели от типа nxm, rxr и mxn, където N=nxr. Ако е изпълнено условието m≥2n-1 мрежата е неблокираща. Частният случай на тази мрежа при m=n=r се нарича мрежа "Мемфис".



Фиг.12-10 Мрежа на Клос

5.4. Мрежа на Бенес

Мрежата на Бенес с N входа и N изхода има симетрична структура - фиг.12-11, 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 $2\log_2 N-1$, а общия брой на КЕ в мрежата е $N(2\log_2 N-1)/2$.



Фиг.12-11. Мрежа на Бенес при N=8.

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.

Литература

1. Амамия М., Танака Ю.
Архитектура ЭВМ и искустенный интеллект., М., "Мир", 1993 г., стр.268-277.
2. Хокни Р., Джессхоуп К.
Параллельные ЭВМ.,М., "Радио и связь", 1986 г., стр.158-170.
3. Tse-yun Feng.
A Survey of Interconnection Networks, Computer, Dec., 1981, pp.12-27.
4. Б.Боровски, П.Илиева.
Мрежи от вътрешносистемни връзки за паралелни компютърни архитектури. Автоматика, изчислителна техника и автоматизирани системи, 1988, №2, стр. 69-81.
5. Howard J. Siegel.
The Theory Underlying the Partitioning of Permutation Networks, IEEE Transactions on Computers, vol. C-29, No.9, Sept. 1980, pp.791-800.
6. Larry D. Wittie.
Communication Structures for Large Networks of Microcomputers, IEEE Transactions on Computers, vol. C30, No4 April 1981, pp.264-273.
7. Стоян Марков.
Изчислителни системи с висока производителност, С., Техника, 1990, стр.65-90.
8. С.О.Степанян

Коммуникационные сети в многопроцессорных ЭВМ. Автоматика и вычислительная техника, 1987,Но.3,стр.31-43.
9. Goodman J.R., Sequin C.H.
Hypertree: A Multiprocessor interconnection topology. IEEE Transactions on Computers, vol. C30, No 12 December 1981, pp.923-933.
10. Ts. Taslakov. Analysis of Multiprocessor Structures Built with Transputers. Proceedings of Second Workshop on Parallel Computing and Transputer Applications, Varna, 25-26 November, 1993.
11. Network topology
http://en.wikipedia.org/wiki/Network_topology#See_also

ТЕМА 13

АРХИТЕКТУРА НА ПАМЕТТА В ПАРАЛЕЛНИТЕ КОМПЮТРИ

Назначението на паметта в компютъра е изключително ясна – да съхранява информацията и да я предава в бъдещето. На нейната организация дълги години се е отделяло много внимание. Понастоящем съществуват два подхода за организация на паметта – вертикална и хоризонтална.

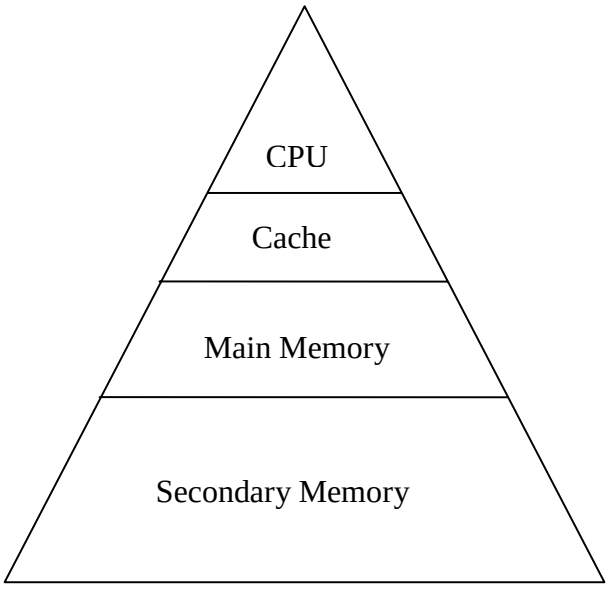
1.Вертикална (йерархична) организация на паметта

1.1 Същност на вертикалната организация

Идеалната структурна организация на паметта е очевидно тази, която е на едно ниво, т.е. процесорът чете и записва данни и команди само от една единствена памет. Такава организация се нарича ”плосък” модел на паметта. Това не е възможно поради следните причини:

- Един бит бърза памет е по скъпа от един бит бавна памет.
- Колкото е по-голям обемът на паметта, толкова по-голямо време е необходимо за достъп до данните, които са запомнени на произволни адреси (при една и съща технология на производство).

Ето защо в почти всички съвременни компютри паметта е структурирана йерархично. Принципът заложен в тази йерархична структура е следния: най-близо до процесора се намира най-бързата, но с малък обем памет, а най-голямата по обем, но и с най-голямо време за достъп памет се намира най-отдалечено от процесора – фиг.13-1:



Фиг.13-1. Структура на паметта

Устройството за обработка в компютри има непосредствена връзка с много малка (с обем от порядъка на няколко стотен думи), но много бърза памет, достъпът до която може да се реализира за един такт. Тази памет се представлява набор от регистри и се явява неотменна част от устройството за обработка (регистрите не са показани на фиг.13-1, защото те са част от процесора).

В компютрите от преди 25-30 години на следващото ниво на йерархия се намира по-голяма памет, наричана оперативна (основна, централна). По време на решаването на задачи в нея се записват основните части на програмата и базата от данни. Поради много голямата разлика в обемите на регистровата памет и основната памет, достигаща до няколко десетки порядъка, а също така и поради значителната разлика във времето за достъп (на няколко порядъка) в съвременните компютри се включва кеш-памет, заемаща междинно място между регистровата памет и основната памет. На най-ниското ниво в йерархията на паметта се намира относително бавната, но с изключително голям капацитет вторична (външна) памет. Съвременното решение е тя да бъде разположена на магнитни дискове, но съществува тенденция реализацията й за в бъдеще да се базира на други технологии – **FRAM** (**F**erroelectric **RAM**), **MRAM** (**M**agnetic **RAM**) или като оптична памет [7].

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.13-1 са обобщени характеристиките на паметите по отделните нива.

Таблица 13-1

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.

Като краен резултат общата стойност на паметта в компютъра е близка до стойността на външната памет, а средното време за достъп малко се различава от времето за достъп на бързите устройства.

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 локалността на обръщенията.

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.

1.2.2. Организация на информацията между различните нива кеш памети.

Типично за кеш паметите е, че обикновено паметта с по-голям номер на нивото съдържа в себе си и информацията, намираща се в паметта с по-малък номер. Например, оперативната памет съдържа в себе си информацията, която се намира в кеш паметта от второ ниво, която от своя страна пък включва в себе си съдържанието на паметта от първо ниво. Общият начин за описание на тази ситуация е да се каже, че L1 (това е информацията съдържаща се в паметта от първо ниво) е подмножество на L2 (това е информацията съдържаща се в паметта от второ ниво). Това е така наречената включваща (обхващаща) (inclusive) структура на кеш паметта. От представянето на серията процесори на DURON на AMD, при x86 процесорите започва да се използва и един друг вариант на организация на кеширането, а именно изключваща (exclusive) организация. Характерното за нея е, че съдържанието на L1 кеш паметта не се копира в L2, а информацията там е различна.

Конкретната причина за това беше, че поради големия обем на L1 (128KB) на DURON и малкия на L2 (64 KB), включващата организация става невъзможна. При изключващата организация обаче L2 има различна информация от L1 и по този начин я допълва. Така на практика при включващата организация общият обем на кеш паметта е равен на обема на последното ниво преди оперативната памет, докато при изключващата организация общият обем на кеш паметта става равен на сбора от обемите на различните нива. Цената за това е известно усложняване на алгоритмите за работа с кеш паметта

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.

За целта се използва спомагателна памет (tag RAM или памет с признаци), в която на всеки фрейм се съпоставя по една част от нея, където се съхраняват признаците за този фрейм. При това положение когато процесорът изисква съответния байт от паметта, първо се претърсва tag RAM, за да се определи дали информацията е налична в кеш паметта или не. Това претърсване обаче може да добави нежелателно закъснение (латентност). За да се намали то, се използват три различни метода на свързване на блоковете от оперативната памет с блоковете от кеш паметта. Това са пълна асоциативност, директна асоциативност и n-кратна асоциативност.

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.

В заключение, може да се направи следното обобщения: кеш с директно съответствие е просто еднопътна асоциативна кеш памет, а напълно асоциативна памет е n-кратно асоциативна кеш памет, където n е равно на общия брой блокове в кеш паметта.

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;

Основната идея е следната. След като се модифицира дадена позиция в кеш паметта например на процесор #1, позициите в другите кеш памети се маркират като невалидни. Сега, ако някой от другите процесори се обърне да прочете операнда от своята кеш памет, ще получи информация за отсъствие на операнда в кеш паметта, което от своя страна влече обръщение към основната памет за извличане на коректната стойност на операнда. В същото това време, процесор #1 записва модифицираната линия от своя кеш в основната памет. Има системи в които е разрешен трансфер между кешовете (с или без едновременен запис в главната памет).

1.2.7. Размер на кешовия блока

С увеличаване на размерът на блока е възможно да се реализират по-добре предимствата на локалността по пространство. Това е вярно до някакъв определен размер. Ако се увеличи размерът на блока, докато размера на кеш паметта остава постоянен, тогава се намалява броят на блоковете, които кеш паметта може да съхрани. По-малко блокове в кеш паметта означава по-малко набори, което влече и по-голяма вероятност от отсъствия на търсената информация. Освен това, ако размерът на блока е много голям, налице е много загубено пространство, защото много блокове ще съдържат само няколко байта от работното множество, докато останалото пространство е неизползваемо. Казано по друг начин, големият размер на блока е по-склонен да “замърсява” кеш паметта с неизползвани данни отколкото малкия размер на блока.

Друг проблем на блока с голям размер е свързан с ширината на лентата на пропускане на шината. При по-голям блок повече данни се извличат с една команда от типа LOAD, големият блоков размер може да “изяжда” лентата на пропускане на шината, особено ако степента на не попадение е висок. Така, компютър с голям блоков размер на кеш паметта трябва да има по-висока ширина на лентата на пропускане, в противен случай латентността ще се увеличава.

1.2.8. Латентност и пропускателна способност

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.

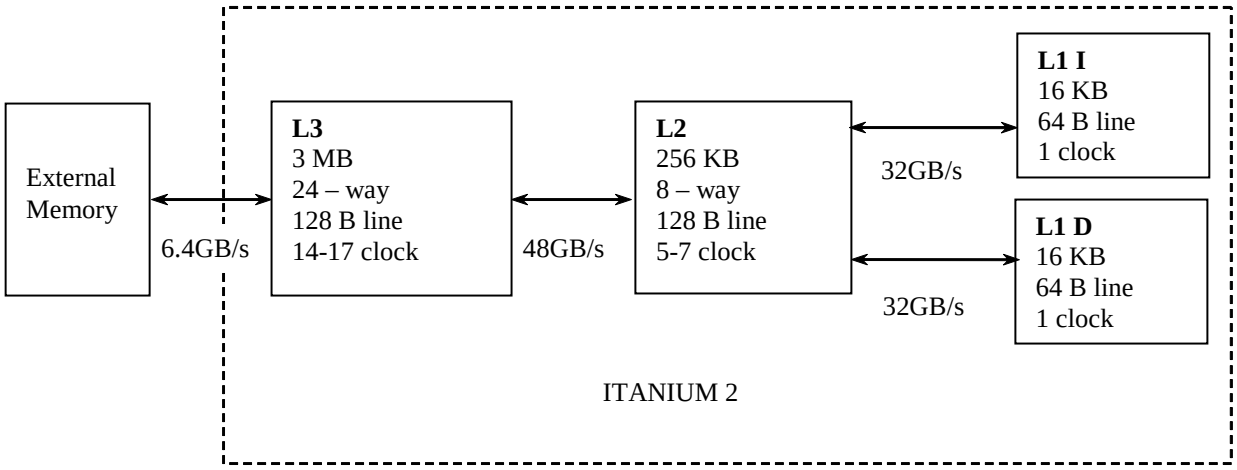
Всички тези времена зависят от конкретната организация на връзката процесор-памет. Обикновено латентността се дава с брой тактови цикли на процесора, които изминават между изпращането на заявката и получаването на отговора. Например, ако процесорът работи на 1 GHz, а тактовата честота на шината е 100MHz, 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 ѝ 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.

И на края, като пример, на фиг.13-2 е показана организацията на кеш паметта в един съвременен процесор (Itanium 2).



Фиг.13-2. 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.

2. Хоризонтална организация [1]

При тази организация на паметта, целта не е да се намали времето за достъп, както при вертикалната организация, а да се увеличи броят на достъпите за единица време. Резултатът в крайна сметка е един и същ – по-бърз достъп до информацията. Тази организация засяга само основната памет и се прилага когато въвеждането на кеш памет е невъзможно или нецелесъобразно, напр.

във векторните компютри и компютрите с потоково управление (виж теми 7 и 11).

Съществуват различни начини за хоризонтална организация, най-простият от който се състои във физическо разбиване на паметта на две половини, достъпът до който може да бъде осъществен едновременно. Поместването на всички команди в едната половина, а данните в другата, увеличава средната скорост на достъп към паметта. Това по същество е Харвардската архитектура на компютъра и за първи път е реализиран в компютъра STRECH на IBM. Понастоящем, в голяма част от процесорите, така е организиран достъпът до информацията записана в кеш паметта от първо ниво.

По-нататъшното развитие на тази идея води до използването на няколко модула памети, свързани към независими процесори с помощта на високоскоростна комуникационна мрежа (виж тема 12), така че всички модули на паметта са еднакво достъпни за всички процесори.

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

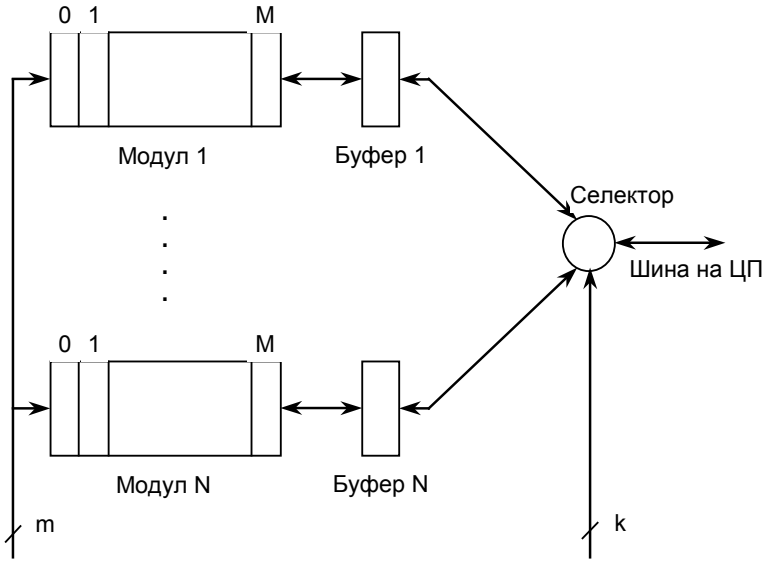
- пакетна обработка на множество достъпи към паметта;
- конвейерна обработка на множество достъпи към паметта.

На фиг.13-3 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 използуват 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.

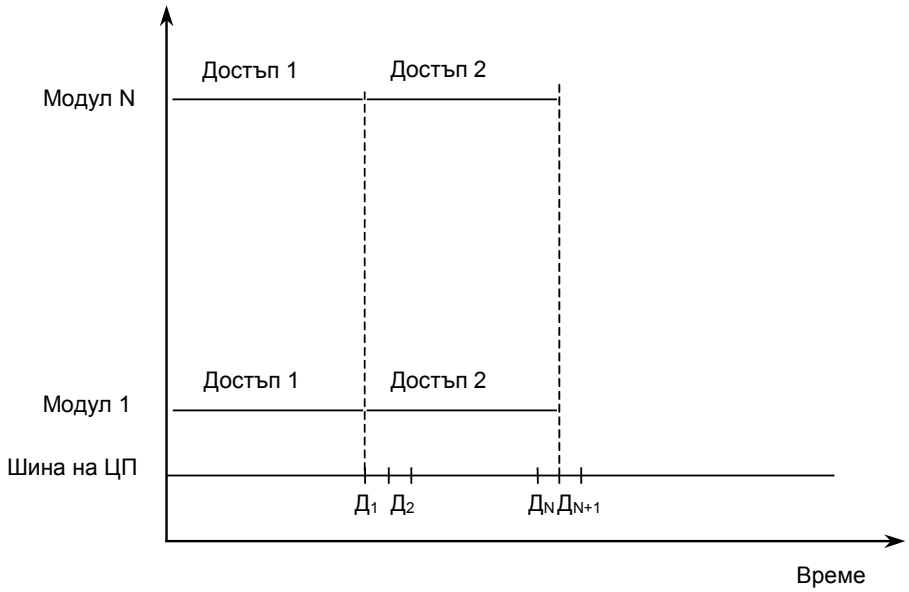
На фиг.13-4 е дадена времедиаграмата за четене на група от думи (диаграмата за запис изглежда аналогична).

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 $\mathbf{q \le N}$ и **t** при $\mathbf{q > N}$ (**t** е времето за достъп до произволен модул).

За да се преодолее, макар и частично, посочения недостатък е предложена друга структурна организация на паметта, известна като конвейерна памет - фиг.13-5. Тук фиксаторите са поставени на адресните шини. Това позволява да се използува за всеки модул свой относителен адрес, отличен от останалите. За да се оперира с тази по-гъвкава структура се въвежда контролер на паметта. На



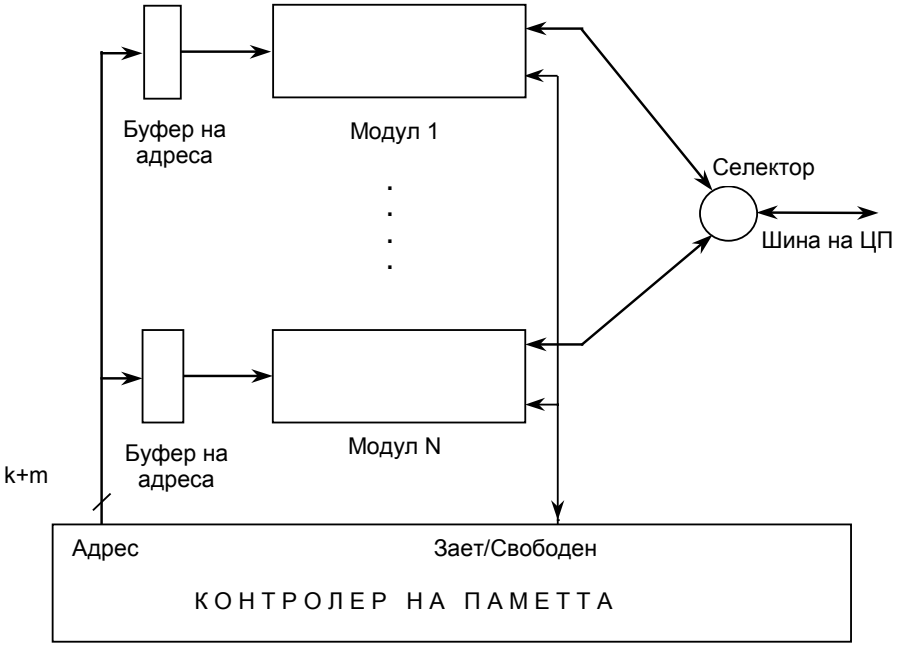
Фиг.13-3. Пакетен достъп до паметта с използуване на редуване на адресите



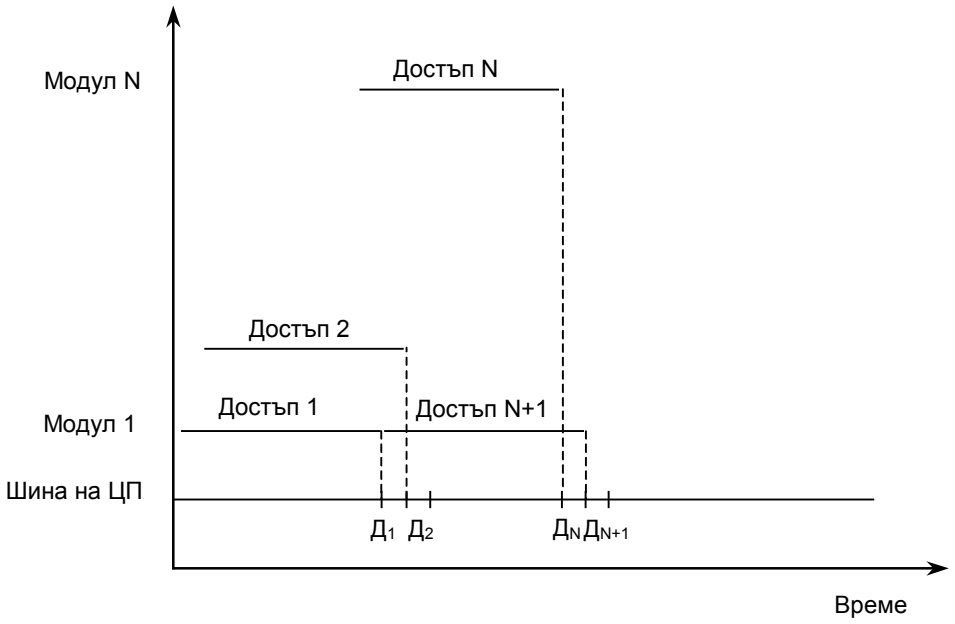
Фиг.13-4. Диаграма за четене на N думи в пакет

това устройство постъпват заявките от конвейерите; всяка заявка изисква достъп до една дума, и тези заявки се обработват последователно. За всяка заявка за достъп, контролерът определя зает ли е в дадения момент модула, съдържащ необходимата дума и ако не е зает, то съответния адрес се съхранява в буфера за адреси и операцията започва. Ако модулът е зает, то заявката се

задържа. Докато модулът памет се обръща към съответната клетка, контролерът по аналогичен начин обработва останалите заявки. Когато модулът завърши операцията, контролерът или стартира нова операция (ако старата е била операция за запис в паметта), или насочва данните от модула към заявилият ги конвейер (за операция четене), преди да се започне нова операция. На фиг.13-6 е показана времедиаграмата за четене (за запис диаграмата е аналогична).



Фиг.13-5. Разслоена памет с конвейерна обработка на заявките.



Фиг.13-6. 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.13-4 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.

3. Разположение на данните в паметта

В тема 8 бяха дискутирани някои от начините за разположение на данните в паметта на компютрите, в зависимост от начина на обработка – паралелно по битове и последователно по думи или обратно. Също така беше посочена необходимостта от ъглово завъртане за преобразуване на форматите на данните, специфични за

двата начина на разположение в паметта. По-надолу се обсъжда един проблем, който е свързан с хоризонталната организация на паметта.

Както стана ясно, разслоената памет е ефективна при подходящо обръщение към модулите ѝ. 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.13-7 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.13-7. 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 използува така нареченото скосено (skewed) разположение – фиг.13-8.

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.13-8. 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.13-7, а операторът SKEWED подрежда данните по начин показан на фиг.13-8.

Така е възможно безконфликтна обработка по редове и по стълбове, но не и по диагоналите. Цялостното решение на проблема е показано на фиг.13-9.

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.13-9. Напълно безконфликтен достъп до елементите на X.

Тук се използуват 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).

Литература

1. П.М.Коуги. Архитектура конвейерных ЭВМ, М., "Радио и связь", 1985, стр.51-62.
2. Амамия М., Танака Ю. Архитектура ЭВМ искустенный интеллект., М., "Мир", 1993 г., стр.227-288.
3. Рональд Левайн, Суперкомпьютеры. В сборник статей Современный компьютер. М., "Мир", 1986, стр. 9-29.
4. Таслаков Ц., Ангелов М. Ръководство за лабораторни упражнения по дисциплината "Компютърни архитектури", Печатна база при ТУ-Варна, 2000.
5. Computer Architecture: The Anatomy of Modern Processors http://ciips.ee.uwa.edu.au/~morris/Courses/CA406/cache_coh.html
6. Кеширане, латентност и пропускателна способност. Хардуер, 1, 2003, стр.26-31.
7. Иванов С., Таслаков Цв. Енергонезависими памети – функциониране, характеристики, възможности. , Електроника'2004, 21-22 май 2004, София.
8. Stephen R. Wheat Intel Itanium 2 Processor: Architecture Overview and Computing for Performance. http://www.sharcnet.ca/fw2003/slides/StephenWheat_Intel.pdf#search='stephen%20r.%20wheat

ТЕМА 14

АРХИТЕКТУРА НА ДИСКОВАТА ПАМЕТ

1. Въведение

Дисковата памет е част от входно-изходната подсистема на съвременните компютри. Същевременно, чрез дисковата памет основно се изгражда вторичната памет, чиято наличност се обуславя от невъзможността да се изгради полупроводникова памет с достатъчно голям обем и време на достъп съизмеримо с такта на шината. Мястото на дисковата памет в йерархичната организация на паметта е най-отдалечено (в логическо отношение) от централния процесор (виж тема 13).

Разлики в производителностите на процесорите и дисковата памет съществуват и продължават да се увеличават. Така например, производителността на процесорите се удвоява всяка година, плътността на паметите – на всеки две години, а на плътността на записа – на всеки три години.

Същевременно много автори считат, че производителността на входно-изходната подсистема е ограничаващ фактор за получаване на висока обща производителност на компютъра. Налага се мнението, че параметрите на дисковата подсистема оказват съществено влияние върху общата производителност на компютъра и усъвършенстването на дисковата памет е също толкова значимо, колкото и това на процесора.

Развитието и усъвършенстването на дисковата памет се извършва в два аспекта: технологичен и архитектурен. Естествено, основно внимание тука ще отделим на архитектурния аспект. Предварително трябва да се посочи, че той не е алтернативен на технологичния, а в известна степен е независим от него и го допълва.

2. Характеристика на работното натоварване.

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. Изисква се някакво време за 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 е времето, необходимо за физическото предаване на байтовите информация между диска и компютъра. Това време се явява функция от броя предадени байтове.

Така времето за достъп и времето за изчакване са независими от времето за предаване. Ясно е, че ако данните са структурирани в големи блокове, общото време за обслужване на една заявка към диска слабо ще зависи от фиксираните загуби по време за търсене и изчакване.

Системите за вход/изход с невисока производителност могат да заемат магистралата между компютъра и дисковата подсистема за цялото време на позициониране, въртене и предаване на данните. Ако блокът, който се предава е с неголям размер, времето за предаване съставлява само малка част от пълното време за обслужване и комуникационната среда (най-често шина) не може да бъде използвана ефективно. Затова системите с по-висока производителност осигуряват механизъм за търсене, при който устройството може да дава заповед за самостоятелно освобождаване на шината по време на търсене на зададената пътечка. Преимуществото се състои в това, че многочислените процедури за позициониране могат да се съвместят по време, намалявайки сумарното очаквано време за вход/изход, което позволява по-добре да се използва наличната пропускателна способност на дисковата подсистемата.

За да се осигури на устройството възможност за превключване на шината, подсистемата за вход/изход трябва да поддържа механизъм чувствителен към положението за въртене, тъй като устройството прекъсва контролера в този момент, когато търсеният сектор се намира под главите. Ако шината вече е заета, устройството трябва да изчака цял оборот, преди отново да се опита да изпрати данните. Задръжката, свързана с невъзможността за заемане на шината, се явява един от основните източници за намаляване на производителността на много от съществуващите подсистеми за вход/изход. Това произтича от отсъствието на буферна памет и необходимостта за обслужване на магнитните дискове в реално време. При появяването на тази архитектура (началото на 70-те г.) буферната памет е била твърде скъпа, а потребността от бързодействащ вход/изход не е била така остра поради по-бавните процесори. Но поради драстичното намаляване на цените на електронните паметите от една страна, а от друга увеличената производителност на процесорите се оправдава включването на буферна памет за пътечките, която може да се запълва веднага след позициониране на главата. След това този буфер се явява източник на данните, когато магистралата стане достъпна.

В зависимост от изискванията съществуват различни варианти за използване на високопроизводителни подсистеми за вход/изход. Диапазонът, който те обхващат се простира от

обработката на малко на брой, но големи блокове от данни, които е необходимо да се реализират с минимална задръжка (вход/изход на суперкомпютрите), до голям брой прости задачи, които оперират с малък обем данни (обработка на транзакции). Важна задача при разработката на подсистема за вход/изход е създаването на такива устройства, които да удовлетворяват такива разнообразни изисквания.

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.

Трябва да се подчертае, че операциите за вход/изход при суперкомпютрите почти изцяло се подчинява на последователен механизъм. Обикновено параметрите за изчисление се предават от диска в структури от данни към паметта в големи блокове и резултатите се записват обратно на диска. При такива използвания е необходима висока пропускателна способност и минимално време за изчакване, независимо, че те характеризират ниска производителност. На това контрастира обработката на транзакции, които се характеризират с огромен брой случайни обръщения, с относително неголеми части от работата и изискващи умерено време за изчакване при много висока производителност.

Тъй като системите за обработка на транзакциите изразходват голяма част от времето за обслужване на търсенето и изчакването, технологическите успехи, водещи до снижаване на времето за предаване, няма да оказват особено влияние на производителността на тези системи. От друга страна, в суперкомпютрите, за търсенето на данни и за тяхното предаване се изразходва еднакво време и затова производителността на такива системи се оказва особено чувствителна към всяко усъвършенстване в технологията на производство на дискове.

3. Дискови матрици

3.1. Основни положения

Известни са различни способи позволяващи да се съкрати времето за достъп, да се намали времето за изчакване или да се намали времето за предаване при традиционните дискове [1]. Към тези способи се отнасят: използването на дискове с фиксирани глави, дискове с паралелно предаване на данни, дискове с повишена плътност на записа, твърдотелни дискове, дисков кеш и на края - механизъм за планиране на достъпа до кеш паметта. Снижавайки времето за обслужване на диска, също така се

намалява времето, свързано с обслужване на устройството в опашката.

Алтернатива на "класическите" методи за повишаване на производителността се явява използването на паралелизъм по пътя на обединяването на няколко физически диска в група с организация на тяхната работа аналогична на един логически диск. Прието е тази група от дискове да се нарича дискова матрица. Идеята е била предложена през 1987г. От Д. Петерсон, Гибсон и Кац от университета Беркли, САЩ. Преимуществото на такъв подход се заключава в това, че пропускателната способност на няколко диска може да се използва да обслужва една логическа заявка за вход/изход или може да поддържа няколко едновременни операции за вход/изход. В допълнение трябва да се отбележи, че матрица от дискове може се изгражда с използването на широко разпространена дискова технология, вместо използването на специализирани и скъпоструващи методи, напр. дискове с фиксирани глави.

Както и за всеки метод, при който данните се разпределят между многото физически устройства, основно слабо място се оказва устойчивостта на откази, т.е. съществува тенденция с увеличение на броя дискове да нараства интензивността на отказите. От тази гледна точка се разглеждат нови възможности при организация на матрици съставени от голям брой твърди дискове, чиято цел е повишаване на устойчивостта на откази при поддържане на висока пропускателна способност за операциите четене/запис и висок процент ефективно използване на капацитета на диска.

За съжаление, надеждността на матриците от дискове (както и на всяко подобно устройство) пада при увеличаване броят на устройствата в матрицата. 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.

Основният подход се заключава в разбиването на матрицата на групи, така че за всяка група да съответстват допълнителни,

"проверяващи" дискове, съдържащи резервна информация. 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.

3.2. Нива на RAID

Съществуват множество способи (нива) за обединение на дискове **RAID.** Всяко ниво представлява от само себе си компромис между пропускателната способност на вход/изхода и капацитета на диска, предназначен за съхранение на информация

с излишък. В оригиналната статия на Петерсон, Гибсон и Кац са описани 5 различни нива – от 1 до 5 на архитектурни решения. Към тези нива в последствие са добавени нови нива, в някои случаи представляващи комбинации от известни решения, както и ниво 0, което е матрица от дискове без излишък

3.2.1. Прости (основни) RAID нива

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 Error Correcting Code**). **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 малка част от

дисковете с данни. Например, при G=10 диска, $C = \lceil \log_2 10 \rceil = 4$ и 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%.

Ясно е, че тази организация използва по-малко на брой контролни дискове спрямо **RAID 1** и по този показател я превъзхожда.

Тъй като единицата за предаване на информация е сектора, използването на побитовото разслоение на дисковете означава, че данните предавани в **RAID 2** трябва да се състоят от **G** сектора. При предаване на неголеми блокове от данни производителността се губи, защото трябва да се записват или четат групите изцяло, независимо от конкретните потребности. Ето защо **RAID 2** се предпочита за суперкомпютрите, но не е подходяща за обработка на транзакции. При записи на големи масиви от данни, тази организация има същата производителност както и **RAID 1**

Понеже всички **SCSI** дискове поддържат възможност за откриване и коригиране на грешки, това ниво в съвременните компютри е малко използвано и е изместено от другите нива на организация.

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

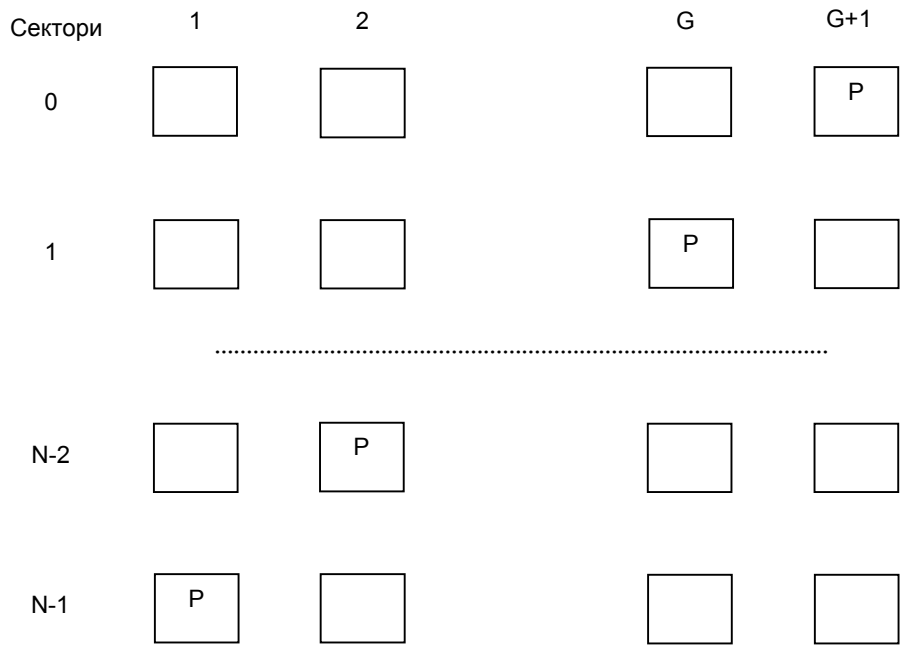
новото значение на четността са необходими стария блок за данни, стария блок за четност и новия блок за данни, или:

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.14-1.



Фиг. 14-1. **RAID 5**- Разпределен контрол по четност.

Един стълб от фиг.14-1 представлява диска, а редовете представляват секторите; блоковете за четност **P** са разположени по-такъв начин, че един блок да принадлежи на един ред и един стълб. Това неголямо изменение оказва огромно влияние на производителността при запис на неголеми масиви от информация. Ако операциите за запис могат да бъдат планирани, така че обръщенията за данните и съответстващите им блокове за четност

да са към различни дискове, се появява възможност за паралелно изпълнение на (G+1)/2 записа.

Тази организация има еднакво висока производителност при запис и четене както при големи, така и при малки обеми от информация, която я прави подходяща в случаите за смесено използване.

***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.

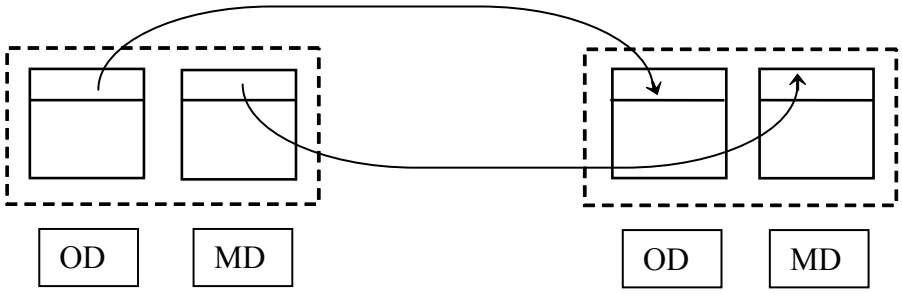
3.2.2. Съставни RAID нива

Разгледаните по-горе нива на дисковете матрици имат своите предимства и недостатъци, ето защо много от тях се използват в различни части от пазара за удовлетворяване на различни приложни изисквания. Резонния въпрос е могат ли да се комбинират някои от нивата за да се вземат техните

предимствата? Тези нови **RAID** нива се наричат различно – *съставни, вместени* **RAID** нива. Не всички комбинации от нива са възможни; някои комбинации са по-често използвани от други. Най-често се комбинира ниво 0 с нива 1, 3 или 5 за да се съчетаят предимствата на ниво 0 – високата производителност с устойчивостта на откази, характерна за другите нива. Тука накратко ще се разгледа комбинирането на ниво 0 и ниво 1. Това е най-популярното ниво от съставните **RAID** нива. **RAID 01** и **RAID 10** (или **RAID 0+1** и **RAID 1+0)** комбинират най-добрите страни – висока производителност в повечето използвания и изключителна устойчивостта на откази без да е необходимо изчисляване на контролни суми. На фиг.14-2а е показана организацията **RAID 0+1**, а на фиг.14-2б - **RAID 1+0** на примера на две групи дискове, всяка от които съдържа също два диска.

Първа група дискове Втора група дискове

а) организация **RAID 0+1**



Първа група дискове Втора група дискове

б) организация **RAID1+0**

Фиг.14-2. Разлика в организациите **RAID 1+0 RAID 0+1**
OD –диск съдържащ оригиналните данни; MD диск съдържащ огледалните данни.

RAID 0+1 е огледална конфигурация на два набора с разделяне на данните. Тази организация съдържа две групи дискове, с еднакъв брой, като първата съдържа оригиналните данни, а втората група – огледалните данни. **RAID 1+0** е

разделяне на данните върху няколко огледални набора. Тази организация съдържа в общия случай D/2 групи, всяка от които с два диска – единия за оригиналните данни, а другия за огледалните данни. С поевтиняването на дисковете, двете конфигурации получават увеличена популярност. Минимум са необходими 4 диска за изграждане на коя да е конфигурация. Като предимство на **RAID 1+0** може да се посочи, че тя осигурява по-добра отказоустойчивост и възможност за възстановяване отколкото **RAID 0+1.**

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.

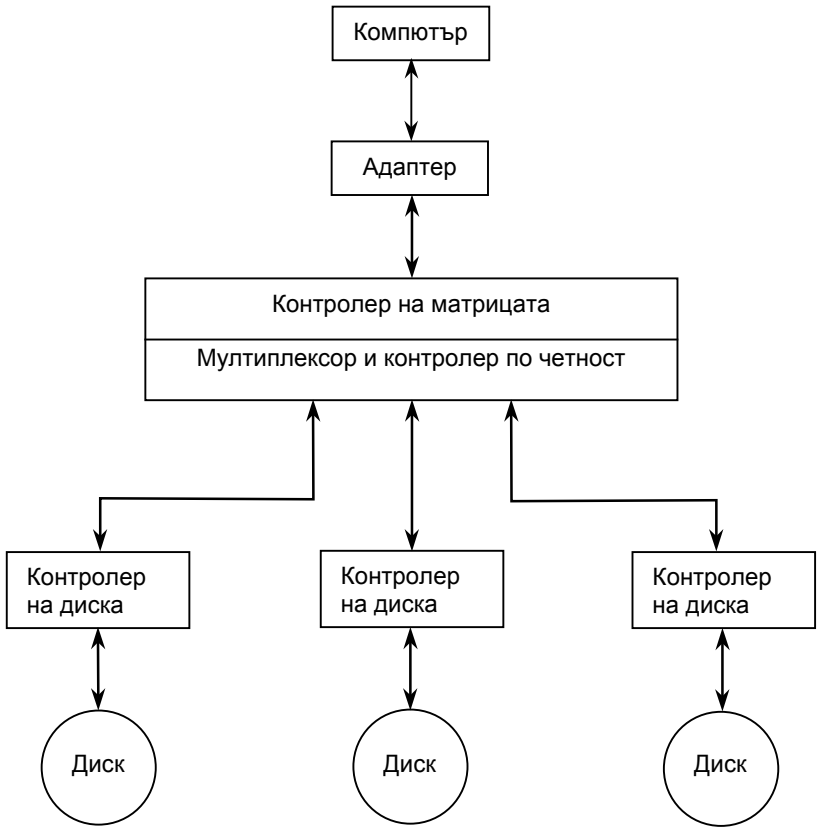
Всички апаратни **RAID** системи имат някакъв вид конфигурационни инструменти, позволяващи създаването на логически информационни единици вътре в устройството. Почти всички поддържат **RAID 0**, **1**, **0+1** и **5** с различни нива на контрол в съответствие как вътрешните дискове са разпределени върху различните логически информационни единици.

4. Архитектура на контролерите.

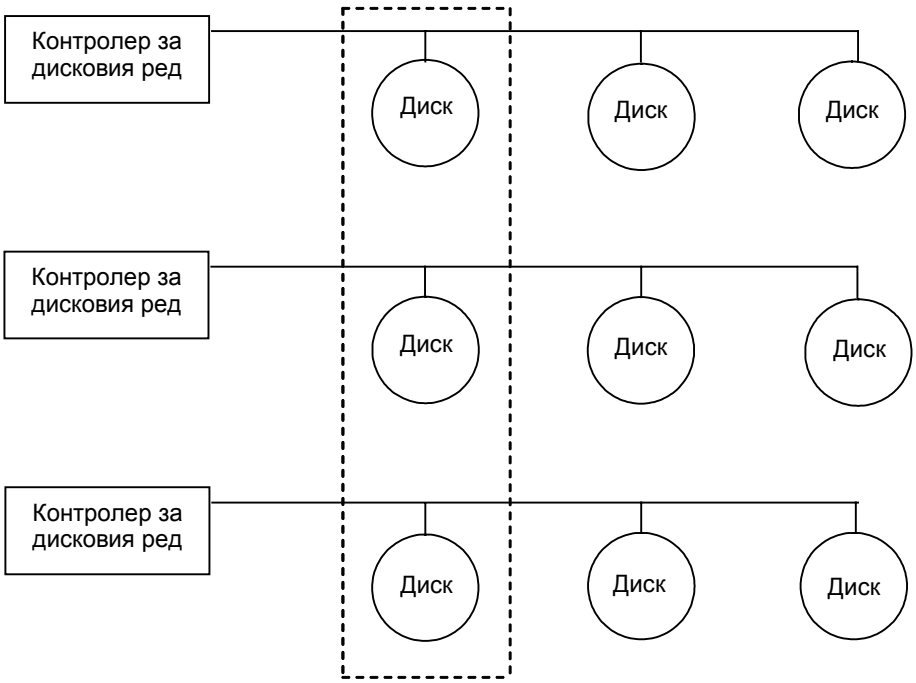
За да функционира дисковата матрица е необходимо не само да се обединят няколко диска, а и да се промени структурната организация и на контролерите. Основното техническо отличие между контролерите за вход/изход на дисковата матрица и по-традиционната архитектура на външната памет се състои в широкото използване на вградени контролери, изпълнени на **VLSI**, а така също и буфери по продължение на комуникационната среда процесор-диск.

На фиг.14-3 са показани елементите на проста матрична подсистема за вход/изход.

Шината между процесорът и дисковите устройства се състои от адаптер на шината, контролер на матрицата и контролер на диска. Интерфейсът между процесорът и адаптера обикновено представлява стандартен канал; интерфейсът между контролера на матрицата и контролера на диска се явява дисков протокол от високо ниво, напр. **SCSI**. Като правило в състава на контролера влизат схемата за корекция на грешките и буферът за пътечките. Адаптерът на основния процесор управлява интерфейса между компютъра и дисковата матрица и обикновено използва за предаване данни между оперативната памет и устройството за вход/изход метода на прекия достъп до паметта. Контролерът на матрицата отговаря за осигуряването на интерфейса с контролерите на отделните дискове и включва такива елементи като буфери на паметта и схеми за изчисление на четността за възстановяване на данните на отказалите дискове. Обаче, най-важната функция на контролера на матрицата се явява установяването на съответствие между логическите и физическите сектори на диска, например реализация за разместване на разслоени сектори за контрол по четност в **RAID 5** и презареждане на възстановени сектори на диска за горещ резерв на матрицата. Контролерът на диска реализира управление на устройството на физическо ниво. Той осигурява логически интерфейс с контролера на матрицата и отговаря за откриване и коригиране на грешки, и обикновено включва в себе си буфер за пътечките, който може да бъде използван за съгласуване на скоростите. Също така той може да осъществява синхронизация на позициониращите механизми, ако трябва да се организира матрица



Фиг.14-3 Компоненти на контролера на дисковата матрица.



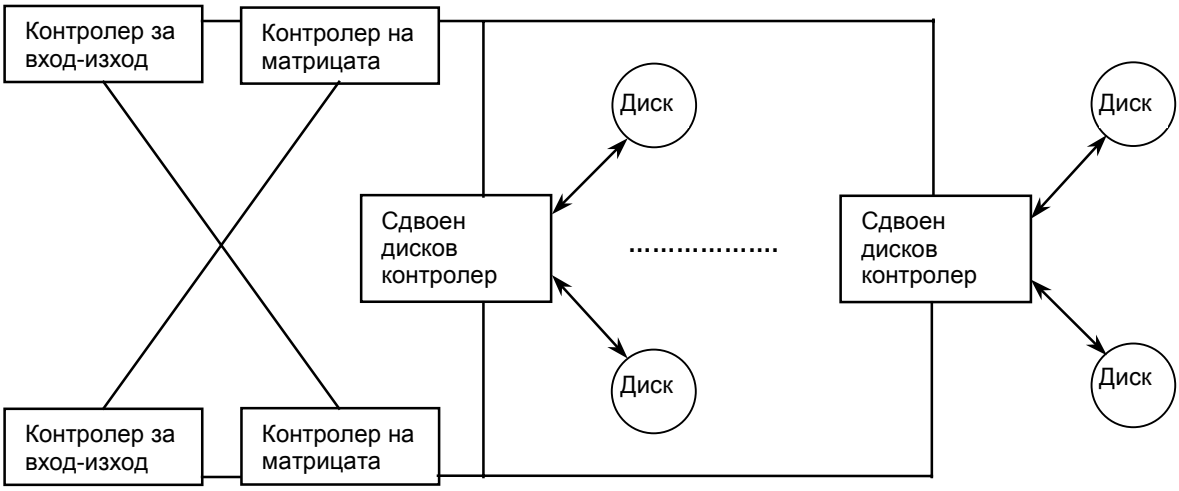
Фиг.14-4. Двумерна организация на матрицата

със синхронни дискове. Шината между контролера на матрицата и контролера на диска може да бъде отделна високоскоростна мултиплексна шина или отделна шина за всеки контролер на диска. Организацията показана на фиг.14-3 може да се разглежда за вход/изход на къси операции като индивидуални дискове, а за вход/изход на големи масиви като една група дискове. Един или повече диска трябва да бъде отделен за горещ резерв за замяна на друг диск, в случай на отказ на последния.

Показаната по-горе организация може да бъде доразвита – фиг.14-4, където контролерът на единичния диск е заменен с контролер на линия (редове) от дискове, а отделното дисково устройство е заменено с ред от дискове.

Дисковете, отнасящи се към една колона, образуват група. Произволен диск може да бъде избран индивидуално или като част от групата. Тази организация се характеризира с увеличена пропускателна способност, която за да се реализира пълноценно може да изисква по-висока пропускателна способност от интерфейса.

В архитектурите, разгледани по-горе, основно е била реализацията на високопроизводителен вход/изход благодарение на използването на допълнителна (в смисъл повече) контролна информация за четност и изключително висока готовност на магнитната среда. На фиг.14-5 е показана една от най-устойчивостта на откази конфигурация на дискова матрица.



Фиг.14-5. Елементи на устойчивостта на откази матрица от дискове.

Цялото оборудване на контролерите и комуникационната среда е дублирано, което гарантира невъзможност за загуба на данни от диска при възникване на единичен отказ в произволна точка на системата. Организацията на такава система е най-подходяща за обработка на транзакции, където е важна високата готовност. Дублирането дава възможност за поддържане на по-

високи скорости за вход/изход, отколкото традиционното решение – с една шина и един контролер.

5. Архитектура на дисковата памет в компютрите с разпределена памет [2,4].

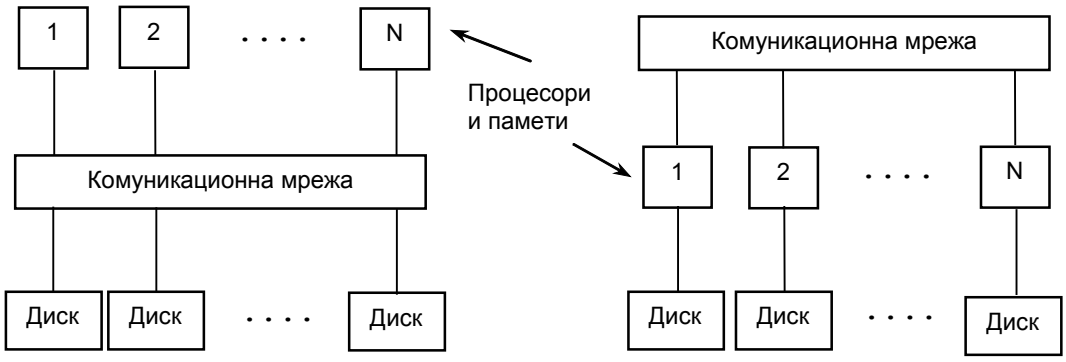
Разгледаната в предходните точки архитектура на дисковата памет е ориентирана към компютри с обща памет. При компютрите с разпределена памет, архитектурата на подсистемата за вход/изход се различава значително от тази на компютрите с обща памет, поради разлики в общата организация на изчислителния процес. Тези разлики водят до разпределена входно/изходна подсистема, която се състои от единични входно/изходни възли, поддържащи едно или повече устройства от вторичната памет.

В обсъждането на входно/изходната подсистема за разпределена памет се предлага следния обобщен модел. Приема се фиксиран брой на процесорите в изчислителната област и фиксиран брой дискове. Всеки диск е асоцииран към входно/изходен възел чрез свързващ контролер. Всеки входно/изходен възел действа като част от разпределен файлов сървър и оперира като посредник между набора дискове и изчислителната област.

По аналогия на начина на организация и свързване на оперативната памет в паралелните компютри и при свързването на външната памет са възможни два подхода – фиг. 14-6. При силно свързаната система – фиг.14-6а, всеки процесор от изчислителната среда има присъединен набор от дискове. Тази система се отличава с твърде малко време за прехвърляне между процесора и локалния диск и висока обща ширина на лентата на пропускане. Единствено големият брой процесори, включени в паралелната система, може да е ограничаващ фактор за изграждане на произволна по размери вторична памет. В системите LCAP-1 LCAP-2 на IBM се реализират разпределени дискове памети по такъв начин.

В противоположност на този подход при изграждане на слабо свързани системи от дискове – фиг.14-6б расте времето за прехвърляне на данни към диска и намалява общата ширина на лентата на пропускане. Възможни са две практически реализации в този случай:

- Входно-изходните възли са разположени по границата (границите) на изчислителната област.
- Входно-изходните възли са разположени вътре в изчислителната област.



а) силно свързана система

б) слабо свързана система

Фиг. 14-6. Обобщени структури на дисковата подсистема на компютрите с разпределена памет.

Пример за реализация на първия подход е архитектурата на Intel Touchstone Delta. В този компютър изчислителната област се състои от двумерна матрица с размери 16x32 изчислителни възела и 16 входно/изходни възела върху всяка страна на матрицата. Към всеки входно-изходен възел са свързани два диска.

Като частен случай може да се разглежда включването на дисковата памет към управляващия процесор. Очевидно паралелният компютър в този случай разполага с един входно/изходен възел и дисковата памет не може да се разглежда като разпределена. Тази организация е подходяща за системи с малък брой процесори и изискванията са същите, както към стандартните компютърни системи.

Intel Paragon е пример за реализация на втория подход. Използвани са отделни входно/изходни възли, но те са разположени на места вътре в изчислителната матрица. Всеки диск реализира **RAID 3**.

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.

В случая е важно да се отбележи, че за осигуряване на необходимата обща производителност на входно/изходната подсистема, с намаляване на броя на входно/изходните възли е необходимо да расте собствената производителност на всеки входно-изходен възел.

Литература

1. Randy H. Katz, Garth A. Gibson David A. Patterson
Disk System Architectures for High Performance Computing. Proceeding of the IEEE, Vol. 77, No 12, 1989.
2. Ц.Таслаков, С.Иванов.
Архитектура на дисковата памет в съвременните компютри. "Автоматика и информатика", 6, 1995,стр.37-45.
3. Момчилова В.
Твърди дискове. PC World, 10, 2000, стр. 32-39.
4. Juen Miguel del Rosario, Alok N. Choudhary.
High Performance I/O for Massively Parallel Computers: Problems and Prospects. – IEEE Computer, 27, March 1994, No3, 59-68.
5. Марков, Ст.
Изчислителни системи с висока производителност. С., Техника, 1990.
6. Твърди дискове. PC World, 4, 2001, стр. 28-34.
7. Слави Славов
Софтуерният RAID срещу хардуерния RAID. "Хардуер", брой 3, март/април 2003, стр.50-53.
8. Интернет адреси посветени на RAID организация
http://www.acnc.com/04_01_00.html

http://www.staff.uni-mainz.de/neuffer/scsi/what_is_raid.html

http://www.storagereview.com/guide2000/ref/hdd/perf/raid/levels/singleLevel4.html