Българска академия на науките

Институт по информационни
и комуникационни технологии

Многопроцесорни архитектури с изчислителни ресурси в основната памет

Д И С Е Р Т А Ц И Я

за присъждане на образователна и научна степен „Доктор“

на ас. инж. Светослав Марианов Ташев

Професионално направление 5.3 „Комуникационна и компютърна техника“
Научна специалност „Компютърни системи, комплекси и мрежи“

Научен ръководител:
Проф. д-р Владимир Димитров Лазаров

София
2015

Благодарности

Първо и преди всичко, искам да изкажа моите благодарности към моя научен ръководител проф. д-р Владимир Лазаров за цялостната му подкрепа в дългогодишното ми професионално и образователно развитие. Оценявам високо всички получени насоки, идеи, мъдрост и насърчения за реализирането на приложения труд. Предоставените ми възможности, отделено време и материална база бяха повече от това, на което всеки студент може да се надява през времето на дългогодишните изследвания за придобитите резултати. Неговата водеща роля е основен инструмент за концепцията на проекта, който предоставя чудесна инфраструктура за изследвания в областта на многопроцесорните архитектури. Проф. д-р Владимир Лазаров е отличен пример за подражание чрез своята отдаденост към научните изследвания и високото качеството на работа.

Също така искам да изкажа специални благодарности на целия екип на Института по паралелна обработка на информацията (сега част от ИИКТ) при Българската академия на науките. Изследванията и развитието в областта на компютърните архитектури изискват изключителна посветеност, много усилия и постоянна работа в екип. Имах щастието да работя с голям брой специалисти, лидери в областта на разработването на компютърни ядра и архитектури. Вдъхновението за много от идеите в тази теза идва точно от колегите в института, от тяхната готовност и ентусиазма, с който откликваха на всички въпроси и проблеми, които възникваха по време на тестовете.

Искам да благодаря и на моето семейство, което постоянно ме подкрепяше и мотивираше по време на дългите ми години работа.

Авторски права

Съдържание

Въведение

В днешно време приемаме компютъра като електронно програмируемо устройство, което се използва за извличане, пренос, обработка и запис на данни. Данните се обработват автоматично в зависимост от набор аритметични, логически или специални инструкции. Възможността този набор от инструкции да бъде променян позволява компютърът да бъде ползван за различни приложения.

Компютърът се състои от поне едно изчислително устройство, еднороден или разнороден вид памет и входно-изходни устройства, които ползваме за пренос на данните до и от паметта. Паметта на една компютърна система може да разделим на няколко основни части: вътрешна, основна, вторична и допълнителна [1]. При повечето съвременни компютри изчислителното устройство, управляващото устройство и вътрешната част от паметта са обединени в общ компонент, наричан централен процесор. Централният процесор има директен достъп до основната системна памет, като този тип памет позволява данните да бъдат четени и записвани с една и съща скорост независимо от реда на достъп [17].

В.1 Развитие на компютрите

В периода от 1939 година до 1941 година Джон Атанасов и Клифърд Бери разработват компютъра ABC Atanasoff-Berry Computer. ABC е първият електронен цифров непрограмируем компютър. Компютърът на Джон Атанасов е удостоен с титлата Milestone на IEEE през 1990 година. За смяна на инструкциите, които са изпълнявани върху входящите данни, е било необходимо ръчно пренастройване на самата машина. През периода на Втората световна война усилията са съсредоточени върху декодирането на криптираните съобщения между воюващите държави. Така през периода от 1943 година до 1944 година Макс Нюман и неговите колеги разработват първата програмируема електронно-цифрова машина Colossus [89].

С изобретяването на транзисторите през 1947 година вакуумните лампи са заменени и започва ерата на „второто поколение“ компютри. Сравнени с вакуумните лампи, транзисторите са по-малки, по-бързи, по-надеждни и изискващи значително по-малко електроенергия. Всичките предимства на транзисторите, както и заниженото топлоотделяне позволяват да бъде произведен един общ полупроводников цокъл, наричан още „централна схема“ или „чип“, съдържащ множество транзистори. Първоначално интегралните схеми са ползвани за създаването на отделни логически елементи, но с

напредването на технологиите интегралните схеми започват да включват все повече транзистори, като в днешно време бройката достига няколко милиарда. Тези нови открития довеждат до изобретяването и разработката на микропроцесора.

В.2 Структура на компютърната система

Компютърът като устройство се ползва в различни приложения за разрешаване на широк набор от задачи [4]. Процесът на разрешаване на един проблем от компютър се разделя на последователни етапи, като всеки етап е последващо ниво от предишния етап. Нивата от съществуващ проблем до разрешаването му чрез компютър са нива на трансформация.

Problem	- Проблем за разрешаване
Algorithm	- Алгоритъм
Language	- Програмен език
Runtime System	- Среда за изпълнение
ISA Instruction Set Architecture	- Архитектура на система от инструкции
Microarchitecture	- Микроархитектура
Logic Gates	- Логически елементи
Circuits	- Физически елементи (схеми)

- Първото ниво е описанието на проблем, който искаме да разрешим чрез използването на компютър. Описанието на проблема правим чрез естествен език. Проблемът при ползването на естествения език е неяснотата на задачата и често срещаното двусмислие при повечето естествени езици [5].

- Алгоритъм – първата стъпка на трансформацията е преобразуването на проблема, обяснен чрез естествен език, в алгоритъм. Алгоритъмът е процедура, описваща стъпка по стъпка проблема и неговото разрешение. Всяка стъпка е прецизно и еднозначно описана, отстраняваща двусмислия и неясноти. За всеки проблем обикновено имаме множество различни методи за разрешаване, които можем да опишем чрез алгоритми. За един метод може да са необходими малък брой стъпки за осъществяване, изпълнявани последователно. Друг алгоритъм може да позволява различните стъпки за решаване на проблема да бъдат изпълнявани паралелно. Алгоритъм, който позволява изпълнението на различни стъпки паралелно, обикновено разрешава проблема по-бързо, въпреки че е по-вероятно борят на стъпките да е по-голям [1].

- Програмен език – следващата стъпка е да трансформираме избрания алгоритъм в компютърна програма чрез програмен език. В днешно време имаме над хиляда програмни езика. Някои езици са универсални, а някои са създадени за конкретни задачи. Пример за специализиран език е Fortran, създаден за решението на формули и ползван за научни

цели. С времето Fortran се усъвършенства и се превръща в универсален език. Друг пример за специализиран език е COBOL, създаден предимно за бизнес ориентирани проблеми. Спрямо хардуера езиците за програмиране може да класифицираме в два типа – от високо и ниско ниво. Езиците от високо ниво са хардуерно независими, като кодът, написан на език от високо ниво, може да бъде изпълнен на различни компютри. Езиците от ниско ниво са силно зависими от избрания компютър за изпълнение.

- Средата за изпълнение (Runtime System) подсигурява софтуерните и хардуерните ресурси, необходими на програмата, за да бъде изпълнена. Средата за изпълнение е съвкупност от ресурси, предназначени да осигурят изпълнението на програмата независимо от ползвания език на програмиране [60].

- Архитектура на система от инструкции (ISA – Instruction Set Architecture) – написаната програма трябва да бъде преведена на машинен език, разбираем за компютъра, който ще бъде ползван за изпълнението на тази програма. Системата команди е набор от специфични инструкции, които компютърът може да изпълни, както и данните, необходими за всяка операция. Преводът от компютърна програма, написана на език от високо ниво, към машинен език се изпълнява от компилатор. Преводът от компютърна програма, написана на език от ниско ниво, към машинен език се изпълнява от асемблер. В това ниво на трансформация определяме типа на данните, адреса на данните и операциите. Организацията на физическата памет, адресното пространство, виртуалната памет, както и приоритетът за изпълнение на различни задачи са също части от архитектурата на системата от инструкции. При многопроцесорна или многоядрена архитектура се определя и как да се изпълняват задачите паралелно [10].

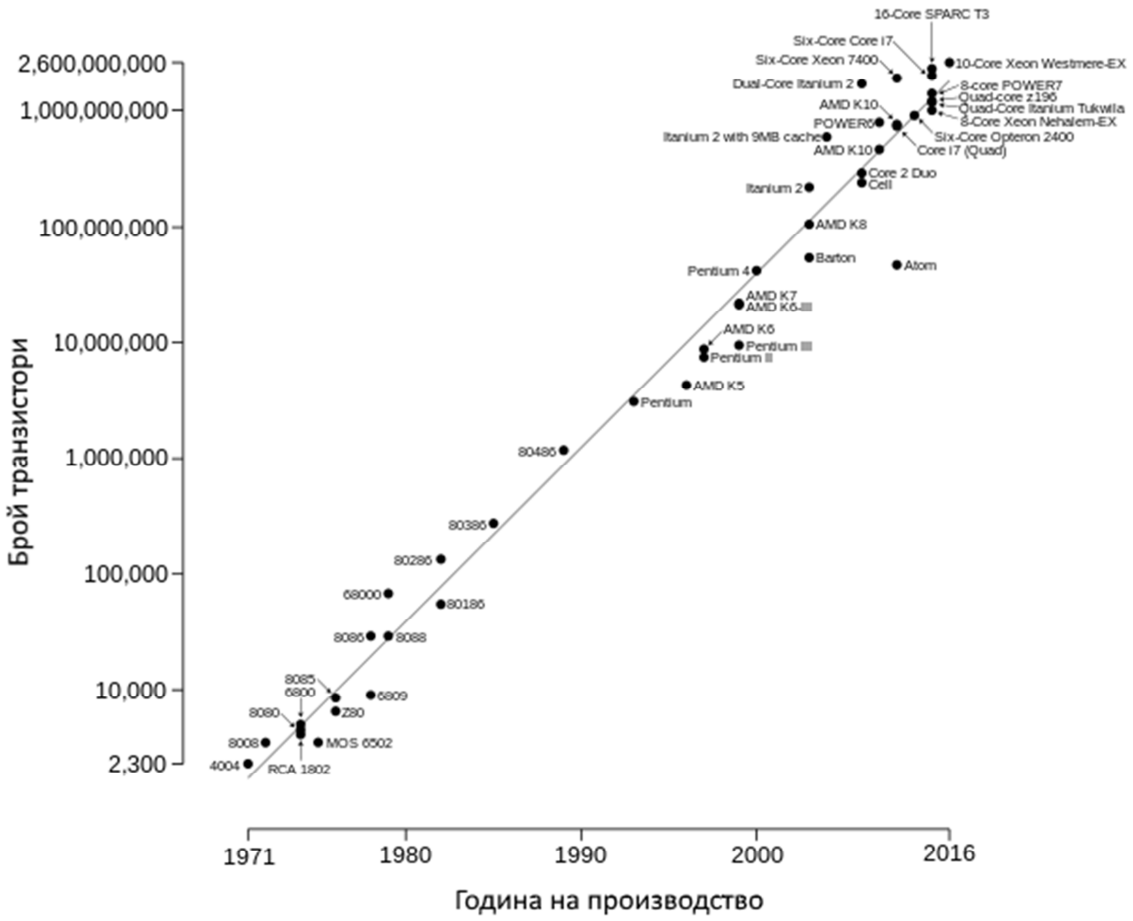
- Микроархитектура – в това понятие се включва начинът, по който набор от инструкции се реализира в процесора. Всяко различно изпълнение дава възможност на дизайнера да разработи различна микроархитектура за различните цели и видове приложения. Имаме възможност да балансираме между цената за изпълнение и производителността на компютъра. Всяка промяна в хардуерната част на архитектурата, която е невидима за софтуера, попада в това ниво на трансформация. Микроархитектурата обикновено се представя като диаграма, която описва взаимовръзките на различните елементи, като това могат да са единични елементи и да се стигне до аритметични логически устройства [41].

- Логически елементи – следващата стъпка е да изградим микроархитектурата от логически елементи. Това са електронни елементи, реализиращи елементарни логически функции като: и, или, не, и-не и така нататък. Тук също имаме възможност за избор на елементи, като изпълнението ще е с различна скорост и производството ще е на различна цена.

- Физически елементи – самите логически устройства са изградени от единични полупроводникови елементи, които се използват за изграждането на транзистори и логическите елементи, разгледани по-горе.

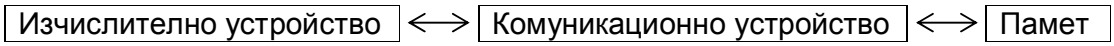
За разрешаването на един проблем, описан на естествен език чрез физически електронни елементи, са необходими много трансформации. Всяко ниво ни предлага много варианти за трансформация, които трябва да бъдат внимателно разгледани, и от избора на дизайнера ще зависи както цената, така и производителността на компютъра.

От различните нива на трансформация виждаме, че хардуерната разработка на компютъра преминава от микроархитектурата до полупроводниковите елементи. С развитието на технологиите транзисторите стават все по-малки и по-ефективни. Гордън Мур, един от основателите на Intel, описва наблюденията си върху развитието на транзисторите. Той отбелязва, че на всеки две години броят на транзисторите, поставени върху интегрална схема, се удвоява. Уповавайки се на факта, че разработването на нови модели интегрални схеми отнема между осемнадесет до двадесет и четири месеца, Гордън Мур прави предвиждането, че тази тенденция ще продължи и в бъдеще. Това предвиждане става известно като „Закон на Мур“ след неговата публикация през 1965 година. Въпреки че тази тенденция продължава дори след половин век, законът на Мур трябва да се разглежда като наблюдение или предположение, а не като физически или естествен закон [103].

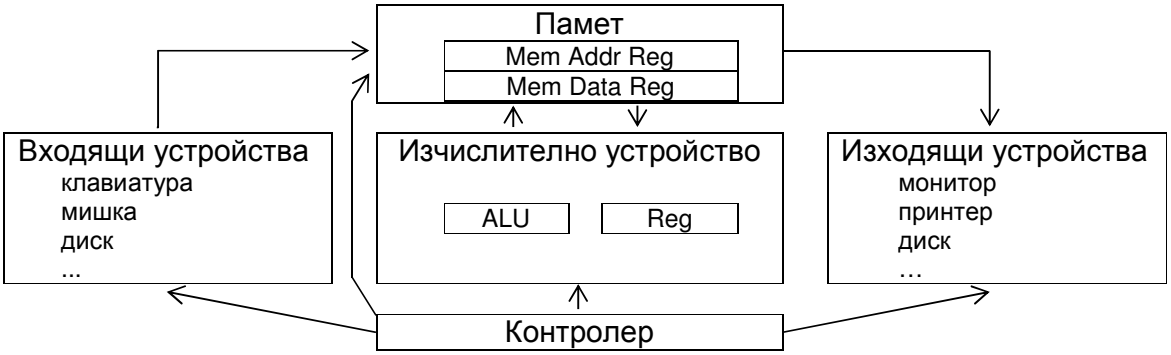


Фиг. В.1 „Закон” на Мур

В основите си една базова компютърна архитектура е изградена от три ключови компонента: изчислително устройство или устройства, комуникационни устройства и памет.



Паметта в една компютърна система е мястото, където съхраняваме програми като поредица от инструкции и техните променливи, както и данни в цифрова форма. Възможността програмата да бъде записана в паметта, за пръв път е била разгледана от Джон фон Нойман.



Фиг. В.2 Модел на Фон Нойман

Инструкциите за обработка на данните, както и самите данни са записани в паметта на компютъра [97]. Ако разгледаме отделна клетка от паметта, няма да знаем дали това са данни, или инструкции. Начинът, по който разграничаваме данните от инструкциите, зависи изцяло от цикъла на достъп до тази клетка. Програмен брояч идентифицира текущата инструкция през последователни цикли. Цикълът за обработка на данни минава през шест етапа:

FETCH INSTRUCTION – извличане на инструкцията – текущата инструкция се извлича от паметта и се записва в регистъра за инструкции;

DECODE – определяне на конкретната инструкция за изпълнение;

EVALUATE – изчисляване на адреса на данните за обработка;

FETCH OPERANDS – извличане на данните от паметта и копиране в регистъра за данни;

EXECUTE – изпълняване на инструкцията, активирана върху извлечените от паметта данни;

STORE – полученият резултат се записва обратно в регистъра за данни или паметта.

Инструкциите може да разделим на три основни класа. Първият клас са операционните инструкции – указват на изчислителното устройство каква аритметична или логическа операция да бъде извършена върху данните. Вторият клас инструкции са инструкции за движение на данните. Третият клас инструкции са за управление на последователността, което позволява контролиран преход към друг ред инструкции.

Освен съхранението на програмите и данните в паметта, друга ключова характеристика на този модел е последователното изпълнение на инструкциите. Това

означава, че в определен момент само една инструкция се обработва. Само една инструкция се извлича, след което се зарежда и едва след като бъде изпълнена върху данните, се преминава към следващата инструкция. Програмният брояч работи като указател на текущата инструкция, посочвайки нейния адрес в паметта. Броячът (указателят) се движи последователно по адресите на инструкциите, с изключение на случаите на специализирана инструкция от класа за управление на последователността. Моделът на Джон фон Нойман се превръща в традиционен метод за проектиране на компютри. Всички доминиращи архитектури на ниво системни инструкции (ISA) и до днес ползват модела на Джон фон Нойман. Това са архитектури като x86, ARM, MIPS, SPARC, Alpha, Power. Архитектурите са с програми, записани в паметта, и последователно изпълнение на инструкциите. На микроархитектурно ниво обаче те са много различни. Нито една от споменатите микроархитектури не следва модела на Джон фон Нойман. Това, към което се стремят разработчиците, е при ISA архитектура да се представи пред програмиста изпълнението на инструкциите последователно, а на микроархитектурно ниво инструкциите да се изпълняват по различен или паралелен метод. С други думи, микроархитектурата изпълнява програмата непоследователно, без това да е видимо за програмиста. Стига микроархитектурата да спазва семантиката, описана на ниво системни инструкции, без да нарушава изпълнението на тези команди, тази идея може да бъде постигната. Въпреки разликите в микроархитектурата това остава прозрачно за софтуера и не влияе на работата на програмиста [85].

Поради невъзможност за достъп едновременно до инструкциите и операндите, а също така и поради разликата в скоростите на процесора и паметта, производителността е много по-малка от скоростта, с която изчислителното устройство би могло да работи. Този проблем е известен още като „Von Neumann bottleneck“. Това сериозно нарушава ефективността на системата, когато изчислителното устройство трябва да изпълнява минимални изчисления върху голям обем от данни. Изчислителното устройство постоянно трябва да изчаква прехвърлянето на данни и инструкции от паметта за обработка.

Новите приложения стават все по-интензивни в обръщенията си към паметта. Обменът на данни, необходим само за едно приложение, се увеличава с добавянето на нови функционалности към него. При паралелните приложения обработката на различни сегменти може да се изпълнява едновременно, което още повече засилва проблема.

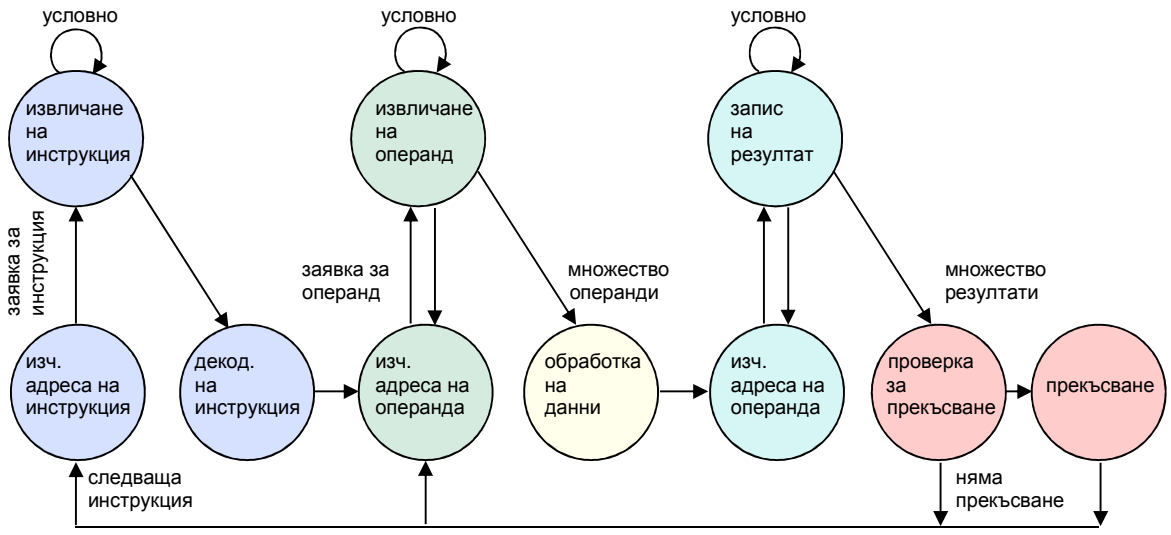
С развитието на процесорите броят на техните ядра се увеличава. Различните ядра опитват едновременен достъп до паметта, което сериозно задълбочава проблема с общите ресурси. Всяко ядро се обръща към паметта през споделена обща шина, допълнително ползвайки обща комуникационна мрежа за обръщения между ядрата. Достъпът до паметта

от различните ядра е неконтролиран процес, което води до смущения на работата на едно ядро от другите. Последиците са: забавяне на едно приложение за сметка на друго, дисбалансирано натоварване на ядрата, непредсказуемо изпълнение на различните приложения в зависимост от паралелно работещите процеси, което прави системата уязвима. Всички тези недостатъци, които ще разгледаме подробно по-долу, налагат необходимостта от нов вид микроархитектура за тяхното преодоляване или облекчаване.

ГЛАВА I. Обзор на взаимодействието между процесора и основната памет и видовете компютърни архитектури

1.1. Взаимодействие между процесора и основната памет

Както разгледахме по-горе, цикълът за обработка на данни минава през шест етапа. Времето за изпълнение на най-бързата инструкция обаче ще отнеме повече от шест процесорни цикъла. Етапът за извличане на данни или инструкции от паметта е условен, в смисъл че зависи от това дали паметта за данните е достъпна, както и шината за достъп до паметта. В допълнение към тази условност, извличането на данни не отнема един машинен цикъл. При добро планиране, в зависимост от местоположението на данните, това може да отнеме средно четири цикъла. Това означава, че изчислителното устройство ще бъде в процес на изчакване през тези цикли [87].



Фиг. 1.1 Цикъл за обработка на данни

Ако разгледаме в детайли изпълнението на различните етапи (Фиг. 1.1) виждаме, че още в първия етап на извличане на инструкции имаме достъп до паметта. Това е условен етап и зависи от състоянието на паметта. Следващите два етапа, декодиране на инструкциите и изчисление на адресите на операндите, са безусловни и отнемат само един машинен цикъл. Извличането на операндите отново изисква достъп до паметта, както и записването на получения резултат. Оттук виждаме, че изпълнението на най-бързата инструкция отнема средно петнадесет машинни цикъла, ако приемем, че общата шина за пренос на данни е винаги свободна и ако паметта е винаги достъпна. Този проблем, както и проблемът с достъпа до общата шина за пренос на данни и инструкции („Von Neumann bottleneck“) налага различни разработки за ефективното използване на изчислителните ресурси.

1.1.1 Структура и ограничения на паметта

Идеалният вариант за изпълнението на една програма при модела на Джон фон Нойман е при постоянен поток от инструкции и променливи. Този поток е без прекъсвания, паметта работи със скоростта на изчислителното устройство, достъпът до паметта е моментален, паметта е евтина, лесна за изработка и имаме достатъчно място за изпълнение и съхранение на всички програми [125]. Различните изисквания към паметта са взаимно несъвместими. Повече памет означава по-бавна памет; бърза памет с кратко време на достъп е обикновено и по-скъпа.

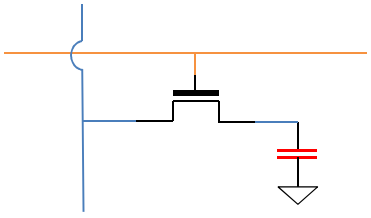
Различните изисквания към паметта налагат разработването на различни видове памет спрямо целите за употреба. От това дали данните се запазват след прекъсване на захранването, може да разделим паметите на два основни типа: енергозависими и енергонезависими. С времето са разработени много различни видове носители на информация. Имаме енергонезависими носители за еднократен запис, както и енергонезависими носители, позволяващи многократни записи. Носителите на еднократен запис са перфокарти, перфоленти, памет на чип, записвана чрез прогаряне на пътечки, оптични дискове и други [18]. Енергонезависимите носители за многократна употреба са статични дискови устройства, твърди дискове, магнитни ленти, магнитни дискове, разновидност на оптичните дискове, флаш памет и други. Всички изброени видове памет са много бавни. Изчислителното устройство на компютъра постоянно изисква данни от паметта. Ако тези данни са недостъпни, то изчислителното устройство буквално спира и ги изчаква, преди да продължи работа. Това налага разработването на памет, работеща близо до или със скоростта на изчислителното устройство. Необходимостта от бърза,

достатъчно голяма и евтина за изработка памет, довежда до разработването на енергозависим тип памет.

При стартирането на съвременните компютри първоначално се зареждат инструкциите от енергонезависима памет за еднократен запис. Това са основните команди за проверка на системата, проверка на паметта, зареждат се основните входно-изходни параметри и се извършва проверка на всички свързани устройства. След тези проверки се зарежда операционната система от енергонезависима памет за многократна употреба. Критичните части от операционната система за работата на компютъра, които се използват най-често от изчислителното устройство, се зареждат в енергозависимата памет на компютъра, която се описва като основна памет. Това позволява непосредствен достъп на изчислителното устройство до тази част от операционната система. При зареждане на програмата за изпълнение или цялата, или критичните части от нея се записват в основната памет. Това се прави отново за непосредствен достъп на изчислителното устройство и бързото изпълнение на тази програма. Когато приключим с изпълнението на програмата, ако имаме резултат, който трябва да се съхрани, записваме резултата в енергонезависимата памет за многократна употреба.

1.1.2 Технология на паметта

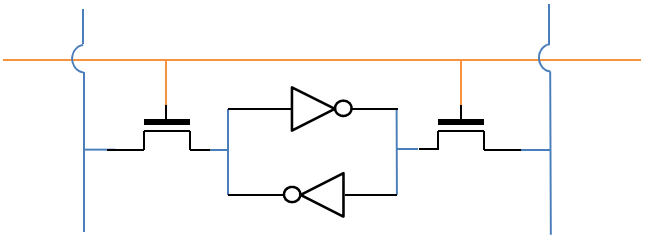
В компютърната система всички данни са представени в числов вид. По тази причина компютърната памет трябва да може да записва единствено и само числа. По редица съображения в работата на компютъра е приета двоичната система. Това ни позволява да разделим паметта на отделни клетки, в които трябва да записваме стойността на числата като единица или нула. Съвременният компютър има милиарди клетки памет и всяка от тях има уникален адрес. Основната памет е памет с произволен достъп или RAM (Random Access Memory). Четенето и записването в различни части от паметта става приблизително за едно и също време. Разработени са два основни вида RAM памет – динамична и статична.



Фиг. 1.2 Технология на DRAM клетка памет

DRAM (Dynamic Random Access Memory) клетката е изградена от кондензатор. Зарядът на кондензатора индикира стойността на тази клетка. Зареден кондензатор определя стойност едно, а кондензатор без заряд определя стойност нула. За да прочетем стойността на клетката, трябва да активираме път към кондензатора през транзистор.

Проблемът при четенето на кондензатора е, че този път за четене не е идеален и имаме оттечки дори когато транзисторът не е активиран. С времето DRAM клетките губят заряд и трябва да бъдат постоянно презареждани. Тази операция по презареждане се извършва от контролера на паметта.



Фиг. 1.3 Технология на SRAM клетка памет

SRAM (Static Random Access Memory) са модел клетки за съхраняване на данни, изградени от два инвертора и два транзистора. Инверторите са последователно едновременно свързани, като по този начин затварят заряда, който им е подаден. Зарядът в клетката ще е постоянен, докато поддържаме инверторите активни. Всеки инвертор е изграден от два транзистора. Когато искаме да прочетем състоянието на клетката, активираме двата транзистора в двата края на инвенторите и така прочитаме разликата в сигналите.

При сравнение на двата вида памет виждаме, че SRAM паметта е по-бърза, но по-скъпа. Изградена е само от полупроводникови компоненти, които са по-бързи. Но SRAM изисква повече транзистори за една клетка, което я прави по-скъпа от DRAM паметта. При DRAM паметта достъпът до кондензатора отнема време, което я прави по-бавна. DRAM клетката е съставена само от два компонента, което я прави по евтина, по-малка и банките памет могат да имат повече клетки. Но DRAM паметта трябва периодично да се презарежда. SRAM клетката е съставена от шест транзистора, което я прави обемна и по-скъпа. Производството на DRAM изисква поставянето на два различни типа компоненти заедно, докато SRAM клетките са изградени изцяло от транзистори.

Когато проектираме банки памет, може да използваме различен вид технология за изграждането на клетките в зависимост от приложението, в което ще ползваме тези банки. Методът на изграждане на стандартните банки памет е чрез единични клетки, подредени двуизмерно от редове и колони. Когато искаме да прочетем клетка от паметта, активираме

реда, до който искаме достъп, след което прочитаме всички клетки от този ред. Стъпките за четене от блок памет са пет:

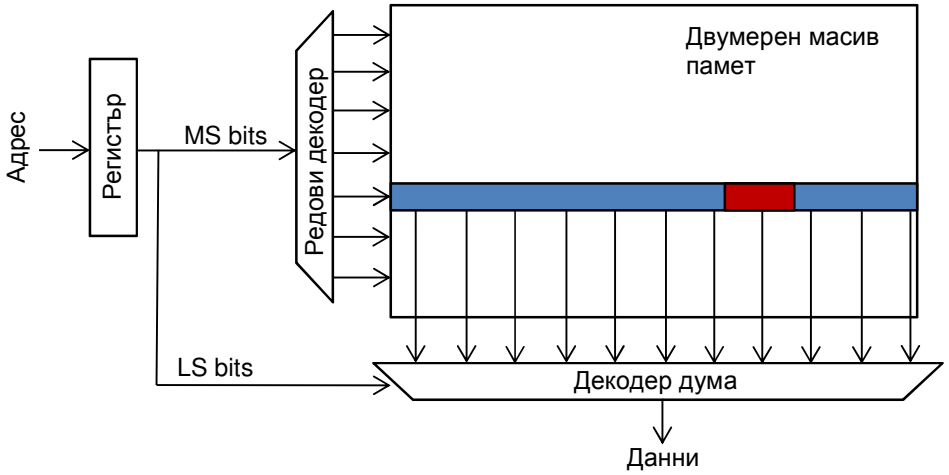
1. Декодиране на адреса, който искаме да прочетем;

2. Активиране на реда спрямо адреса, който ще четем;

3. Избраните клетки активират линиите спрямо техния заряд и целия ред се прочита заедно;

4. Избират се нужните колони спрямо адреса и отчитаме заряда на активираните линии;

5. Презареждане на всички клетки за следващия достъп.



Фиг. 1.4 Организация на банка памет

Времето за достъп до паметта се определя главно от стъпки 2, 3 и 4 – колко бързо може да активираме реда за четене и след това колко бързо клетките ще активират изходните линии [45]. Това определя времето, за еднократен достъп до паметта. Латентността за достъп при повторно четене се определя от цикъла на паметта. За времето на пълен цикъл ще има значение и стъпка 5 – времето, за което презареждаме прочетения ред.

1.1.3 Йерархия на паметта

Въпреки наличието на различни технологии и разнообразие памет не съществува един тип, който да удовлетворява всички изисквания за изграждането на компютърната система. Изискванията за достатъчно голяма памет, бърза и с ниска производствена стойност, не могат да бъдат постигнати с едно ниво памет. Разделянето на паметта на различни нива ни позволява прогресивно нивата да стават по-големи, но по-бавни, без да влияят критично на системата. Предимствата при разделянето на паметта са, че може да

преместим активните части от програмите в по-малки и по-бързи видове памет, близо до изчислителното устройство. Програмите са изградени от цикли, които често правят обръщения към едни и същи данни от паметта [5]. Като цяло може да разделим паметта на четири основни нива:

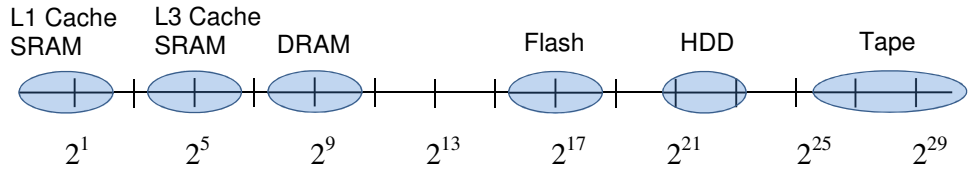
Вътрешна – Регистри и кеш памет;
Основна – Оперативна памет;
Вторична – Локална налична памет, като твърд диск, флаш и други;
Третостепенна – Отдалечена памет, като магнитни ленти, оптични дискове и други.

Общото време за достъп до данни от определено ниво ще зависи от времето за достъп до паметта плюс това дали тези данни са в даденото ниво. В случай че търсените данни не са в търсеното ниво, трябва да се обърнем към следващото ниво в йерархията. Тази зависимост може да изразим със следната формула:

$$T_i = t_i + m_i * T_{i+1}$$

T_i – Общо време за достъп;
t_i – Време за достъп до конкретно ниво;
m_i – Коефициент на непопадение при липсващи данни в конкретното ниво;
T_{i+1} – Общо време за достъп до следващото ниво.

Целта при проектирането е общото време за достъп да е приблизително равно на времето за достъп до конкретното ниво на допустима производствена цена: $T_i \approx t_i$. За тази цел имаме три варианта – да намалим един от следните фактори: t_i, m_i или T_{i+1}. Ако се фокусираме да намалим времето за достъп до нивото t_i, това ще е пряко пропорционално свързано с обема на това ниво: за да е бърза паметта, тя трябва да е с ограничен размер. Друга опция е да намалим достъпа за време до следващото ниво T_{i+1}, което отново ще е пряко свързано с обема на нивото. Най-ефективният метод е да се организира добре движението на информацията между различните нива. Ако преместим фокуса върху намаляване на пропуски при заявки на данни (m_i да е близко до 0), това ще ни позволи следващото ниво да е по-голямо и по-евтино, а текущото ниво по-малко и по-бързо. Тази зависимост налага разделянето на паметта на допълнителни поднива:



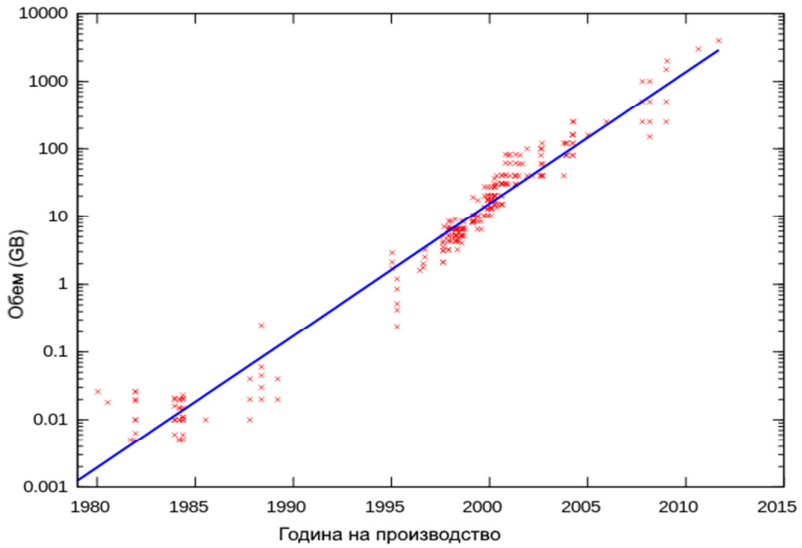
Фиг. 1.5 Време за достъп до различните нива памет в ns
Типично време за достъп при конвенционални процесори, работещи на 4 GHz

- Регистри – регистрите са клетки памет, работещи със скоростта на изчислителното устройство и разположени до него. Освен нуждата от памет, работеща със скоростта на изчислителното устройство, регистрите са необходими заради ограниченията на шината за достъп между изчислителното устройство и паметта. Този вид памет е изключително скъп и това ограничава разработването на модел, в който цялата памет работи до изчислителното устройство с неговата скорост. Основното предимство на регистрите е в случаите, когато работим с данни, които са необходими за повторна обработка. Първоначално регистрите са били ползвани като складове за временно съхранение на данни или инструкции. Впоследствие се разработват регистри за съхранение на адреса за данни от паметта, като по-късно регистрите стават универсални и в тях можем да записваме както данни, така и адреси към паметта.
- Кеш памет – е също част от вътрешната памет на процесора. Всяка структура, използвана за временно съхранение на данни в процесора, чрез която елиминираме достъпа до основната памет, попада в категорията кеш памет. Данните са просто дублирани от основната памет, която е по-бавна, но много по-голяма. Повечето архитектури базират кеш паметта на SRAM технология. Самата кеш памет е разделена на различни нива в различните процесори по същите причини, по които е разделена цялата памет. При многопроцесорните архитектури имаме възможност различните нива кеш памет да бъдат споделени между ядрата или разпределени за ползване от конкретно ядро. Ако решим да изпълним паметта като споделена между ядрата, ще усложним взаимовръзките, както и сложността за изработка. Разпределената кеш памет ще намали обема значително за всяко конкретно ниво. Така обикновено имаме три нива кеш памет: първото ниво е разпределено, второто ниво е споделено между двойки ядра и третото ниво е споделено от всички ядра.
- Основна памет – тя е също енергозависима и бързодостъпна памет, която ползваме за временно съхранение на често използвани данни. Всъщност това е паметта на компютъра във Фон-Нойомановия смисъл. Компромисът, който правим при основната памет е, че заменяме обем за скорост. Големината на паметта увеличава и нейната латентност. Повечето архитектури базират основната памет на DRAM технология. Тази памет е директно или индиректно свързана към процесора чрез шина за данни. В повечето случаи имаме две шини: за адресиране и за данни. Процесорът първоначално изпраща заявка към паметта през шината за адресиране, след което желаните данни се изпращат през шината за данни. След стартиране на компютъра части от приложенията се зареждат от енергонезависимата към основната памет. Устройството за управление на паметта създава обща виртуална памет с виртуални адреси, включваща основната и вторичната памет.

- Вторична памет – или външната памет, се различава от основната по това, че процесорът не може да се обърне директно към нея, както и от това, че е енергонезависима. За достъп до вторичната памет процесорът използва специализирани входно-изходни операции. За вторична памет обикновено се използват твърди дискове или статични дискови устройства. В тази категория попадат също флопи дискове, флаш устройства, някои видове магнитни ленти и оптични дискове. Устройството за управление на паметта използва вторичната памет да записва по-рядко ползваните блокове на една програма в обща виртуална памет. Капацитетът на вторичната памет нараства експоненциално с времето (Фиг. 1.6).

- Третостепенна памет – обикновено се изпълнява с механично включване или изключване към линия за трансфер от робот. Данните често се прехвърлят на вторичната памет, преди да бъдат обработени. Предимно се използва за архивиране или при необичайно големи бази от данни, които се ползват рядко и без намесата на оператор. Когато компютър има нужда от данни върху третостепенната памет, първо се обръща към каталог, за да определи тяхното местоположение. След това инструктира робот, който да вземе и да включи носителя на тези данни към линия за трансфер. Когато приключи извличането на данните, роботът ще изключи носителя от линията за трансфер.

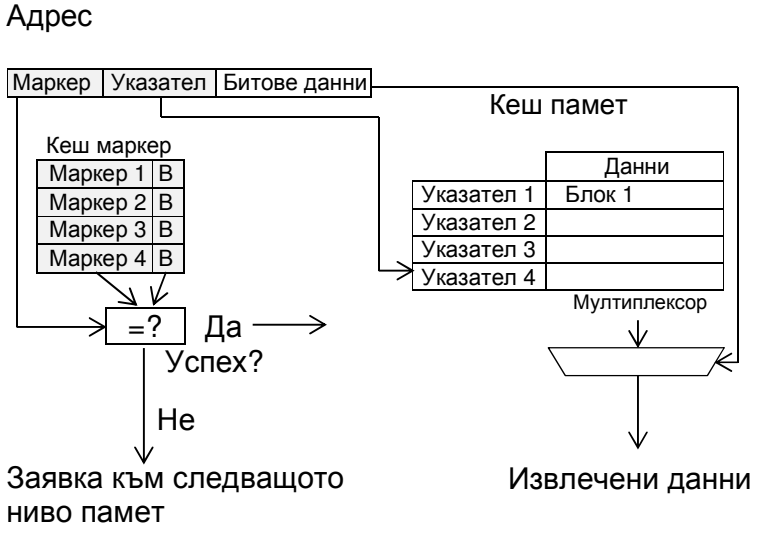
- Отдалечена памет – това е неактивна памет, която не е под управлението на процесора. Данните са записани върху носители от вторичната памет, които са физически изключени от компютъра. Трябва да бъдат включени от оператор, преди да могат да бъдат използвани отново. Този тип памет се ползва за пренасяне на информация или като архив на резервни копия от съществуващи данни. Отдалечената памет увеличава сигурността, тъй като до данните не може да има достъп без човешка намеса.



Фиг. 1.6 Развитие на дисковото пространство във времето

1.1.4 Адресиране

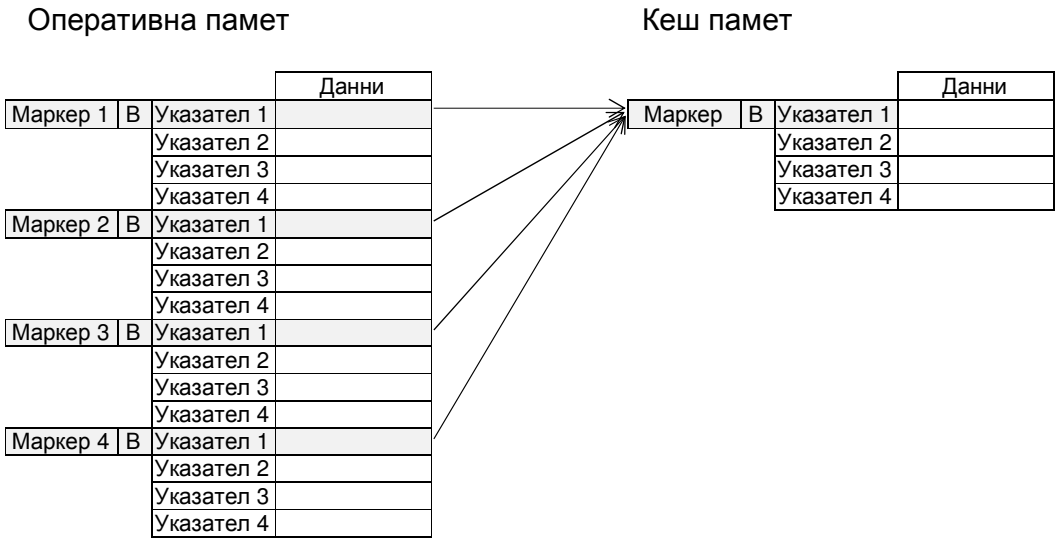
При съвременните процесори имаме две или повече кеш нива. Горните нива са разпределени между ядрата на многоядрените системи, а едно или повече нива са споделени. Когато изчислителното ядро има нужда от конкретни данни, дава заявка към първото ниво на кеш паметта за тези данни. Ако те не са в това ниво, контролерът на кеша дава заявка към следващото ниво от кеш паметта. Стъпката се повтаря, като може контролерът да стигне до заявка за данните от основната памет и тогава тези данни се трансферират към процесора. Кеш и оперативната памет са логически разделени на блокове (линии), които сочат към зони в паметта. Всяка линия или блок на паметта е обозначена с указателен адрес, а група от блокове – с маркер. Блоковете в кеш паметта се зареждат с копия от избрани блокове на оперативната памет, с техните указатели и маркери. Когато изчислителното устройство изисква определени данни, първо проверява дали маркерът на групата присъства в кеша. Ако маркерът на групата присъства в кеша, тогава можем да извършим успешно извличане от кеш паметта, без да се обръщаме към основната памет. Въвеждат се и допълнителни битове, които се ползват за флагове, указващи състоянието на данните, като бит за проверка валидността на данните. При стартирането на компютъра всички битове за валидност се установяват като „невалидни“. Когато изчислителното устройство поиска данни, при съвпадение на маркера на тяхната група се прави проверка и на тяхната валидност, преди да се пренесат данните към регистрите.



*в – бит за проверка валидността на данните

Фиг. 1.7 Извличане на данни от кеш паметта

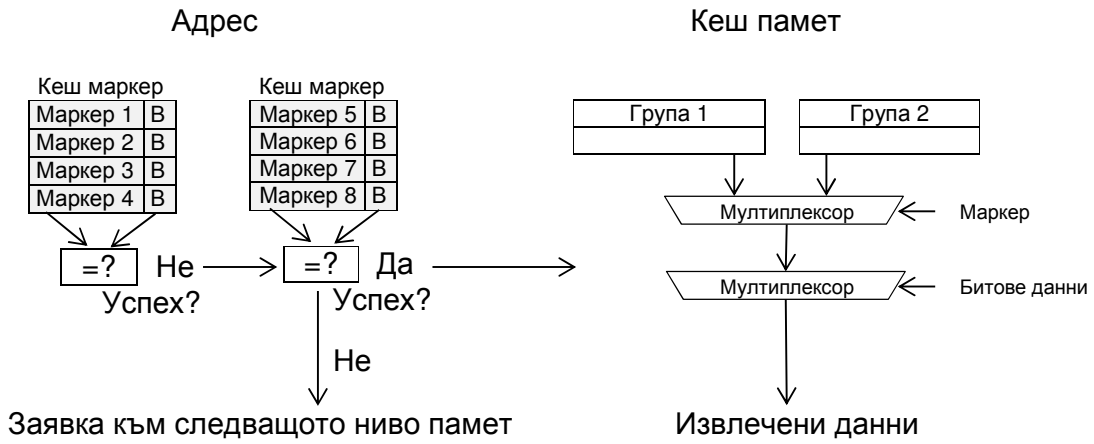
Указателят ни дава адреса на блок в кеш паметта, а чрез маркерите може да разделим група от блокове за записване в кеша. В зависимост от реализацията кеш паметта може да ни позволи запис на различни блокове от оперативната памет или на цялата група, указана чрез техния маркер. Когато една група от последователни блокове с техния маркер може да бъде записан в кеша, имаме „директно съответствие“.



*В – бит за проверка валидността на данните

Фиг. 1.8 Директно съответствие

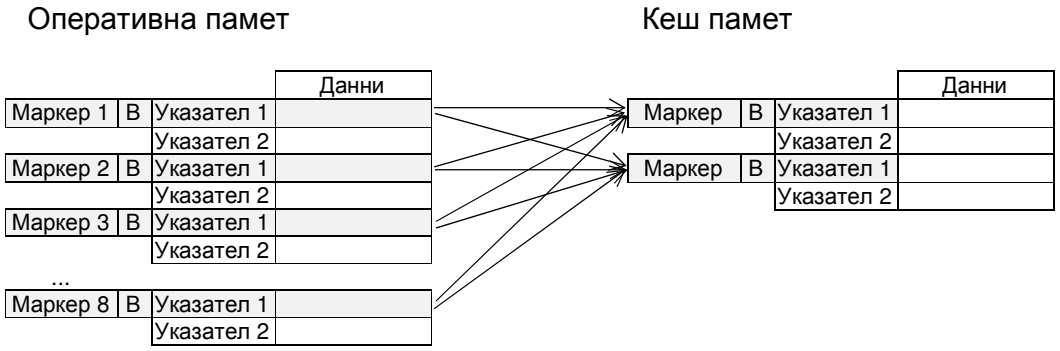
Въпреки че при директното съответствие можем да запишем четири блока в кеша, не можем да запишем гъвкаво който и да е блок от оперативната памет в кеша. Това може да доведе до конфликтни пропуски при четене от кеш паметта в случаи на последователно четене от оперативната памет на блок едно и блок пет. Блок пет и блок едно ще имат еднакви указателни битове, но различни битове за маркер. Всяко прочитане ще бъде пропуснато от кеша и трябва да бъде извиквано от оперативната памет, ако логиката на зареждане в кеша е на база последно прочетени данни. Всеки път ще зареждаме предишната група от данни в кеша с техния маркер. За да намалим пропуските при четене от кеш паметта, може да променим дизайна, като вместо една колона с четири блока, разделим кеша на две колони с по два блока.



*в – бит за проверка валидността на данните

Фиг. 1.9 Извличане на данни при групово асоциативна памет

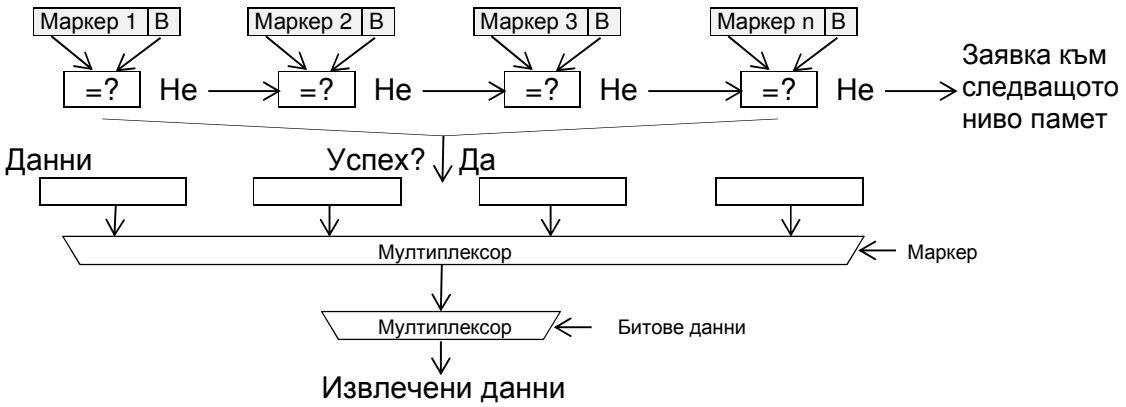
При групово асоциативния модел имаме същия размер кеш, но в него можем да запишем по два блока от различни групи с различни маркери. Указателният ни адрес ще е по-малък, адресирайки само два блока от група. Адресът на маркера ще е по-голям, защото ще имаме по-голям брой групи в оперативната памет.



*в – бит за проверка валидността на данните

Фиг. 1.10 Групово асоциативна кеш памет

Ако разгледаме пак примера с програма, която чете данни от блок едно и блок пет на оперативната памет, ще видим, че в този случай може да заредим двата блока заедно в кеша. Това ще бъде блок с маркер едно и указател едно от първата група и блок с маркер три и указател едно от втората група, заредени заедно в кеш паметта. По този начин ще има по-малко пропуски заради конфликти при заемането на кеша, но ще трябва да се правят допълнителни проверки.



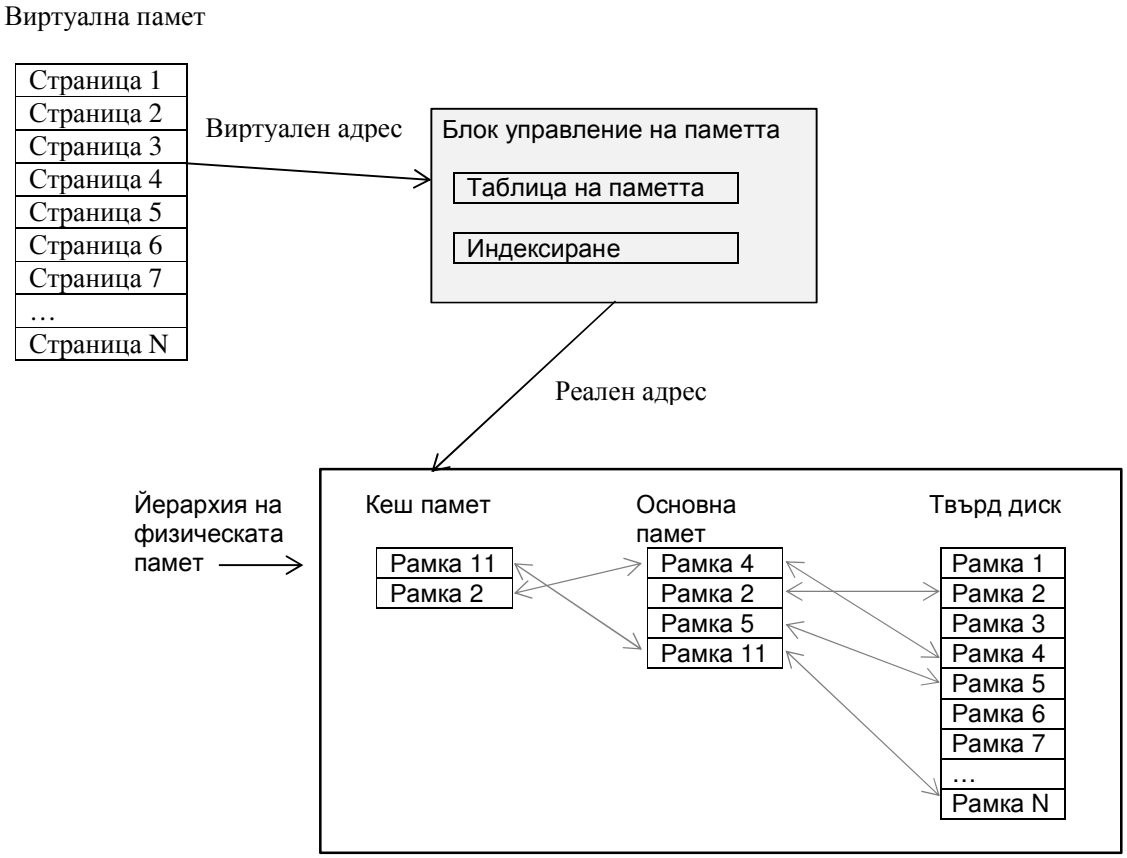
*в – бит за проверка валидността на данните

Фиг. 1.11 Напълно асоциативна кеш памет

По принцип е възможно и напълно асоциативно съответствие, при което блок от оперативната памет може да бъде във всяко местоположение на кеша (n-позиции – Фиг. 1.11). Нямаме указателни битове за сметка на големината на маркерите. Всеки блок от паметта ще притежава уникален маркер. Пълната асоциативност ще увеличи шанса за извличане на данните от кеша, вместо от оперативната памет. За всеки блок от кеша трябва да имаме отделен хардуер за проверка на маркерите на блок паметта. Модулите за проверка, както и големината на мултиплексора за извличане на блока, с успешно сравнен маркер, ще оскъпи напълно асоциативната кеш памет. Всички допълнителни проверки се извършват от допълнителен хардуер, което също така ще увеличи и времето за достъп до паметта. Успеваемостта на четене не е пропорционална на асоциативността на кеша. При стандартни приложения имаме намаляваща възвращаемост към висока асоциативност. Също така при асоциативност, различна от директната кеш памет, изниква въпросът кой от блоковете да заменим при запис на нов блок. Ако някой от блоковете или група от блокове е с невалидни данни, тогава ще заменим невалиден блок. Ако обаче всички блокове са валидни, трябва да задействаме процедура по замяна на блок. Има много различни варианти, като може да заменим последно прочетения блок, най-малко използвания, последно записания или случайно избран блок. Модификация на случайно избран блок, без да включваме последно ползвания за замяна, е често срещан метод и подходящ за по-голямата част от задачите. Различните методи ще са подходящи за различни приложения и трябва да бъдат внимателно избрани при дизайн на системата, защото успеваемостта при достъпа до кеш паметта е критичен за производителността на цялата компютърна система.

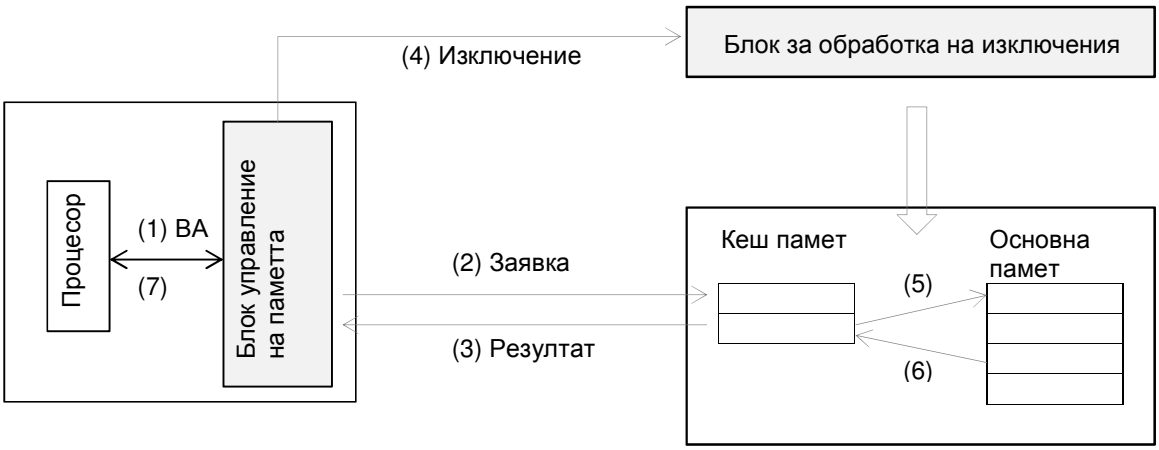
1.1.5 Виртуална памет

От гледна точка на програмиста паметта трябва да изглежда като едно пространство. Илюзията за едно пространство дава свободата на приложението да работи с размера на вторичната памет и скоростта на основната памет. Тази илюзия се постига чрез създаването на виртуално пространство, разделено на страници. Приложението се сегментира с размерите на тези страници, като различните страници могат да бъдат заредени както в основната, така и във вторичната памет. Физическите адреси на паметта се разделят на рамки, като тези рамки имат същия размер, както страниците от виртуалната памет. Страниците от виртуалната памет се записват в различните рамки от физическата памет. Това, че страниците и рамките имат идентичен размер, ни дава свободата да не се грижим за фрагментацията на различните блокове. Таблицата за адресиране притежава съответстващите адреси от виртуалната към физическата памет. Когато едно приложение даде заявка с адрес, този адрес отговаря на виртуален адрес и обръщението е към виртуалната памет. Устройството за управление на паметта преобразува адреса от виртуалната страница към адреса на физическата рамка.



Фиг. 1.12 Адресиране чрез виртуална памет

Ако търсеният адрес не е върху основната памет, а върху вторичната памет, като твърд диск, системата за виртуалната памет записва или прочита страницата от диска и коригира адресирането. По този начин програмите стават независими от конфигурацията на паметта. Създава се илюзията, че разполагаме с цялата физическа памет като основна памет. Компилирана програма може да работи върху машини с различни размери памет, като дори да имаме по-малко основна памет, тогава повече страници ще бъдат заредени във вторичната. Програмите обикновено не използват цялата налична основна памет. Части от програмите, като механизмите за откриване на грешки, се извикват рядко за обработка. Различни сегменти от една програма се обработват в различно време. Страницирането ни позволява да заредим само необходимите страници от много приложения едновременно в основната памет, като с това увеличаваме възможността да обработваме няколко програми едновременно. Когато процесорът извика нова страница, устройството за управление на паметта проверява дали тази страница е вече заредена в основната памет. Ако страницата е заредена в паметта, устройството за управление на паметта изпраща резултата на процесора. Ако страницата е заредена във вторичната памет, тогава устройството за управление на паметта извлича страницата от вторичната памет и я зарежда в основната. Ако физическата основна памет е изцяло заета, тогава операционната система връща някоя от неизползваните страници обратно във вторичната, за да освободи място за новата страница. Операционната система решава коя страница да върне обратно във вторичната памет на базата на това кога страницата е ползвана последно. Всеки изпълняван процес разполага със собствена таблица за съответствие на страниците си. По този начин различните страници на различни процеси могат да бъдат заредени в обща памет, като всеки процес ще ползва толкова памет, колкото му е необходимо. Имаме чисто разделение на паметта между различните процеси. Всеки процес остава с илюзията, че разполага с цялата памет.



(1) Заявка от процесора с виртуален адрес
(2-3) Заявените данни са предварително заредени
(4) Заявените данни липсват от кеша
(5) Отхвърляме избрана рамка от кеша
(6) Записваме липсващата рамка в кеша
(7) Подаваме данните към процесора

Фиг. 1.13 Преход от виртуален към реален адрес със защита на данните

Виртуалната памет ни позволява различни процеси да работят с общи рамки, като имаме възможност да забраним към системните рамки да има достъп от обикновени процеси и обратното. Системни процеси могат да се обръщат към рамките на всички процеси. Към проверката дали една рамка е заредена в основната памет, в таблицата на паметта добавяме контролни битове. Чрез тях указваме дали тази рамка е заета, системна, достъпна за четене, само за запис, обновена или обща. По този начин виртуалната памет служи и за контрол на достъпа до различните данни. Това се постига, като увеличим таблиците за съответствие с допълнителни контролни битове. Имаме възможност да укажем дали може да четем, записваме или изпълняваме страниците от физическия адрес.

Недостатък на виртуалната памет е допълнителното забавяне, което се получава при обръщението към виртуалните страници. Винаги губим време за превод на виртуален адрес към реален. Друг проблем е при системи, използващи статични дискови устройства за вторична памет. Всяка клетка памет на статичните дискови устройства има ограничен брой изтривания и презаписвания. Ползването на такъв диск няма да е подходящо за страниране поради големия брой заявки за записи. Размерът на таблиците за съответствие също трябва да се вземе предвид. Самите таблици трябва да са във физическата памет през цялото време, което ще заеме реално пространство от основната памет. Този проблем се решава чрез въвеждането на специализирана кеш памет за преводи на адреси. В днешни дни имаме операционни системи, като Android, които не ползват виртуална памет. При такива операционни системи, ако един процес не може да заеме необходимата му оперативна памет, обикновено се проваля при изпълнение.

1.2 Архитектури на системните инструкции и микроархитектури

Въпреки че микроархитектурата и архитектурата на системните инструкции са тясно свързани, компютри с различна микроархитектура могат да споделят и изпълняват идентична архитектура на ниво системни команди. Отличен пример за подобни микроархитектури са процесорите Intel Pentium и AMD Athlon, които ползват x86 архитектура, но имат коренно различен вътрешен дизайн (микроархитектура). В основата на архитектурата са самите инструкции и интерфейса „хардуер – софтуер“, който ги изпълнява. В зависимост от това къде е поставен интерфейсът, се определя натоварването на компилатора или хардуера. От нивата на трансформация виждаме, че софтуерът дава заявка за изпълнение на задача, а заявката се конвертира в машинен код в зависимост от микроархитектурата, която ще го изпълни. В основата си машинният код се състои от код на операции и операнди. Кодът на операциите определя какво правят инструкциите, а операндите – върху какво изпълняваме тези инструкции. Това изисква архитектурата да описва системата от операции, с които разполагаме в микроархитектурата.

1.2.1 Архитектурата на системните инструкции (ISA)

През последните десетилетия са разработени много различни видове архитектури на системните инструкции. Фундаменталните разлики между различните видове архитектури се определят от начина, по който е дефиниран интерфейсът между хардуера и софтуера. Може да класифицираме различните архитектури в няколко типа: ЦИСК (CISC – Complex Instruction Set Computing) архитектурата е с голям набор специализирани инструкции. РИСК (RISC – Reduced Instruction Set Computing) архитектурата е с намален брой инструкции. Теоретично архитектури от типа МИСК (MISC – Minimal Instruction Set Computer) минимален брой инструкции и ОИСК (OISC – One Instruction Set Computing) архитектура с една инструкция са разработени, но никога не са използвани в индустриални процесори. Друг вариант е VLIW (VLIW – Very Long Instruction Word) много дълга инструкционна дума: архитектура, при която процесорът паралелно получава няколко различни инструкции за изпълнение, групирани в една дълга инструкция.

Архитектурата на системните инструкции се занимава и с управлението на типа, броя и размера на регистрите. Тук можем да направим избор за начина на работа с данните. Тип архитектура „load/store“ ни позволява данните да бъдат обработвани само чрез първоначален запис в регистри. След обработване на данните от инструкциите резултатът трябва да бъде записан обратно в регистри. Примери за такива архитектури са

MIPS, ARM, както и много други RISC архитектури. Другият тип архитектури са „memory/memory“, при които инструкциите могат да бъдат изпълнени директно върху данни от паметта. Този тип архитектура ни позволява работа както върху данните в регистрите, така и изпълняването на операции директно върху клетки от паметта, без да е необходимо зареждането им в регистри. Примери за такива архитектури са x86, VAX, както и много други CISC архитектури.

За адресиране на паметта най-лесният метод е по абсолютна стойност. Записаният адрес е адресът на паметта, към който искаме да се обърнем без модификации. В зависимост от програмата може да изберем по-бързи методи за достъп до паметта. Един от тях е метод по абсолютна стойност с изместване. Чрез адресиране по абсолютна стойност плюс изчислено изместване имаме възможност да адресираме последователно съседна клетка от паметта, без да е необходимо да зареждаме нейния адрес. Това е много бърз и подходящ метод за системна обработка на масив от данни, който е последователно подреден в паметта. Друг специализиран метод е чрез индиректно адресиране към регистър. Ефективният адрес на данните е зареден в конкретен регистър, който използваме за адреса на данните. Добавянето на различни методи за достъп към паметта ни дава гъвкавост за избор в зависимост от приложението. Използването на допълнителни методи за достъп до паметта има и своите недостатъци. Колкото повече методи добавяме, толкова повече затрудняваме работата на компилатора. Компилаторът ще трябва да избира между различните методи спрямо операциите. Друг недостатък е усложняването на микроархитектурата. Всеки нов метод трябва да бъде осъществен в микроархитектурата на процесора. Алтернатива на машините с регистри са „стек“ машините. При тях данните и инструкциите се записват последователно, линейно в стек. Този метод улеснява хардуерно микроархитектурата, но усложнява много компилацията, особено когато трябва да обработваме много данни едновременно.

Комуникацията с входно-изходни устройства също е част от архитектурата на системните инструкции. Тук имаме основно два типа за комуникация. Първият метод е като заделим част от паметта, запазена за външните устройства. С това адресно пространство те ще могат да оперират, като четат и записват данни, след което програмата, работеща на компютъра, може да обработва записаните данни. Другият метод е с входно-изходни инструкции, активиращи физически канали към външните устройства. Предимството на първия метод е, че е с общо предназначение – позволява ни да добавяме много и различни видове устройства. Недостатъкът на втория метод е ограничението на физическите канали, както и на стандарта, който ползват.

В архитектурата на системните команди попадат и следните елементи: привилегиите за използването на ресурсите – коя програма ще вземе определен ресурс;

процедурите за откриване на грешка или грешна инструкция; процедурите за задействане при заявка от външно устройство; организацията на паметта, задаването на виртуална памет, както и защита на данните. От предложената микроархитектура в дисертацията всички тези елементи от архитектурата ще останат непроменени. Добавянето на нови елементи към микроархитектурата, без съществени промени в архитектурата на системните инструкции, ще ни позволи да ползваме текущите разработки в тази област, като промените остават прозрачни за софтуера от гледна точка на програмиста.

1.2.2 Микроархитектури

Организацията за изпълнението на системните инструкции от хардуера се определя от компютърната микроархитектура. Архитектурата на системните инструкции и микроархитектурата са тясно свързани. Архитектурата на системните инструкции представя програмния модел, като асемблерен език, докато микроархитектурата главно се занимава със структурата на ниско ниво и множество детайли, скрити за програмния модел. Идентична архитектура на системните инструкции може да бъде изпълнена от различни микроархитектури в зависимост от цената на изработка, скоростта на изпълнение или целите на задачата. Микроархитектурата описва частите от процесора, като диаграми, които описват и взаимовръзките на различните машинни елементи. Това може да включва единични логични елементи, регистри, като се стига до комплексни изчислителни ядра, които са представени като един символ в диаграмата. Критичните части при проектирането на микроархитектурата са времето за изпълнение на най-сложната инструкция, времето за изпълнение на най-често срещаните инструкции, както и балансът на използвания хардуер за изпълнение на архитектурата. Целта е да нямаме неизползван хардуер или място, което забавя работата на останалите елементи.

Концепцията на микроархитектурата включва основно четирите етапа от изпълнението на инструкциите (независимо от това дали имаме еднопроцесорна или многопроцесорна архитектура) – извличане на инструкция, декодиране, етап на изпълнение и запис на резултата. Аспект на микроархитектурата е последователността при обработването на командите. Въпреки изискването за последователно изпълнение на инструкциите, имаме възможност за паралелна обработка на етапите на различни инструкции. В различните ресурси на процесора може да изпълняваме различни етапи едновременно, без да нарушаваме последователността на приложението:

- По време на цикъла за декодиране на инструкция едновременно извличаме следващата инструкция;

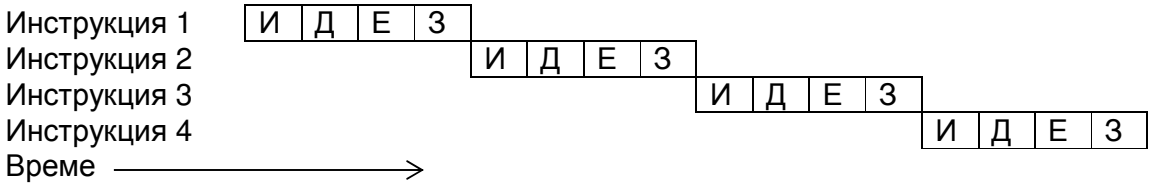
- По време на цикъла за изпълнението на инструкция декодираме извлечената инструкция от предишния цикъл;

- През времето на запис на резултата в регистър изпълняваме следващата инструкция;

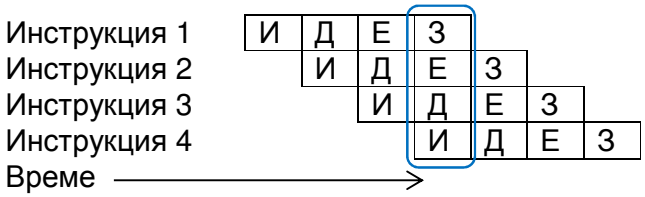
- По време на извличането на инструкция декодираме предишната инструкция.

Всички тези четири етапа от различни инструкции могат да бъдат изпълнени паралелно, стига различните ресурси на процесора да са отделени. Ако маркираме различните етапи с: И – извличане, Д – декодиране, Е – изпълнение, З – запис, то двата модела ще изглеждат по следния начин:

Стандартен многостъпков модел за последователно изпълнение на инструкции



Конвейерно изпълнение на различните етапи от различни инструкции



Фиг. 1.14 Конвейерна обработка

Разделението на процесорния цикъл на различни етапи ни позволява всеки етап на различен цикъл да бъде обработван паралелно, в отделен ресурс на процесора. По този начин запазваме модела на Джон фон Нойман за последователното изпълнение на инструкциите, като същевременно оползотворяваме максимално наличните ни ресурси. Тази конвейерна обработка на инструкциите е известна като „Pipelining“. Можем да наречем системата напълно поточна, ако на всеки такт записваме нова инструкция за обработка.

Недостатъците при конвейерната обработка на инструкциите затрудняват пълната заетост на наличните ни ресурси.

1. Различното време за обработка на различните етапи – това е основен недостатък при конвейерната обработка. Като пример, времето за извличане на инструкция от паметта отнема повече време от изпълнението на инструкцията. Дисбалансът между различните етапи увеличава цялостното време за обработка и завършване на една отделна инструкция.

2. Зависимост между данните – общите операнди между различни инструкции е друг сериозен недостатък при конвейерната обработка. Операндът на една инструкция може да е резултат от предишна. Това е известно като зависимост между данните и при тази ситуация инструкцията не може да се изпълни, преди да завърши записът на предходната. Проверката за зависимост на данните от различни инструкции може да се извърши софтуерно или хардуерно на микроархитектурно ниво, като това налага добавянето на допълнителен хардуер.

3. Празни цикли – това са ситуации, при които след изпълнение на инструкция нямаме нужда от достъп до паметта за запис. Въпреки това при конвейерната обработка имаме етап, предвиден за запис на резултата. Стъпката от този етап за запис ще остане неизползвана, което ще доведе до допълнително забавяне на времето за цялостната обработка на тази инструкция.

4. Споделени ресурси – тъй като паметта и шината за трансфер на данни са споделени от всички инструкции, не можем да записваме резултат, да извличаме променливи или инструкции в един и същи момент, което допълнително затруднява конвейерната обработка. За момента проблемът е облекчен чрез няколко механизма: чрез поставянето на допълнителни регистри, чрез отделни регистри за инструкциите и за операндите и чрез отделни шини за достъп до различните регистри за едновременен достъп до инструкциите и променливите. Всички тези решения са част от микроархитектурата на компютърната система, но оскъпяват и усложняват нейната изработка.

5. Преходи – условните или безусловните преходи са места, на които приложението прескача към изпълнение на друг набор от инструкции. При преход инструкцията за извличане ще е различна от тази, която следва по ред. За преход в приложението няма как да знаем, преди да изпълним инструкцията, указваща прехода. Също така при условен преход няма как да знаем кой ще е следващият набор от инструкции до изпълнение на условието. Когато стигнем до преход, конвейерът ще претърпи забавяне до зареждането на новите инструкции. За частичното разрешаване на този проблем в микроархитектурата на компютъра е разработено хардуерно устройство за прогнозиране на преходите. Това устройство има критична роля за производителността на архитектурата. При условен преход устройството за предвиждане се опитва да предвиди кой ще е следващият набор от инструкции. Предвидените инструкции се извличат и изпълняват спекулативно. Ако по-късно се окаже, че предвижданото е било грешно, изпълнените инструкции се отхвърлят и конвейерът трябва да се презареди.

Като част от справянето с проблема „Von Neumann bottleneck“ в микроархитектурата се въвежда извънредното изпълнение на инструкции. При заявка за данни, които не са заредени в кеш паметта, процесорът изпълнява наличните инструкции,

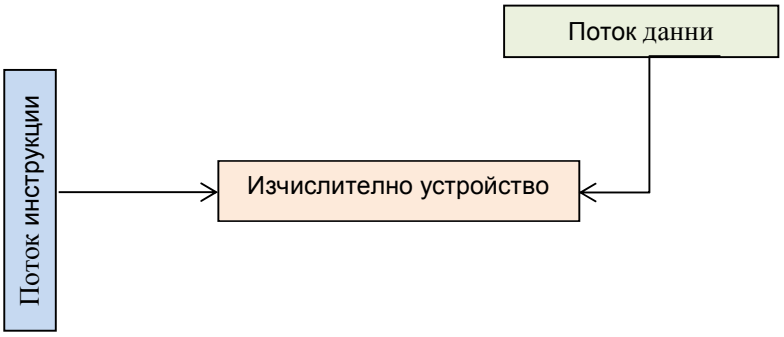
без те да са последователни. Този метод е известен като OOE (Out-of-Order Execution). През времето, в което чакаме данните да бъдат заредени в кеша за изпълнение от изчислителното устройство, изпълняваме извънредно наличните инструкции и техните операнди. Изработването на хардуера за този тип изпълнение е изключително сложно. Резултатите от предварително изпълнените инструкции се записват в буфери. След изпълнение на изчакваните инструкции устройството реорганизира резултатите, така че да изглеждат изпълнени последователно. Буферите също ни позволяват повторно изчисление на инструкциите, в случай на зависимост между операндите или преход от изчакваната инструкция. Това остава скрито за софтуера и програмиста.

1.3 Класификация и видове компютърни архитектури

Компютърната архитектура е комбинация от неговата микроархитектура и архитектурата на системните инструкции. Паралелната или конкурентната обработка на информацията има много форми в компютърната система. Можем да формулираме поток като последователност на обекти от данни или описание на задачи като инструкции. Всеки поток е независим от другия поток и всеки елемент от потока може да съдържа един или повече обекта. В зависимост от начина, по който ги обработваме, Майк Флинн класифицира компютърните системи в четири отделни класа. От двата потока към процесора, данни и инструкции, разделяме категориите като:

- Единична инструкция оперира върху един елемент данни;
- Единична инструкция оперира върху множество от данни;
- Различни инструкции оперират върху един елемент данни;
- Различни инструкции оперират върху множество от данни.

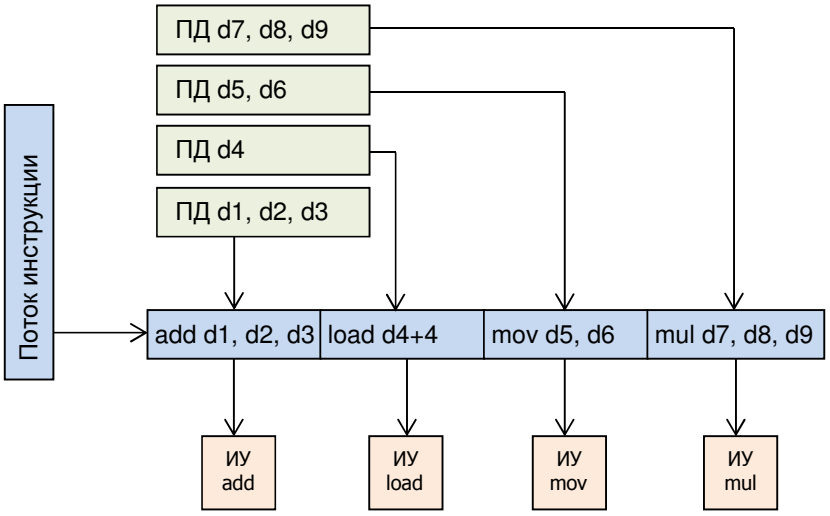
1.3.1 SISD multiprocessing



Фиг. 1.15 Единична инструкция оперира върху един елемент данни

SISD (Single Instruction Single Data) – единична инструкция оперира върху един елемент данни. Това е последователна компютърна архитектура, при която единично изчислително устройство обработва една инструкция във всеки един момент, а резултатът се записва в единична памет. Този модел директно кореспондира с модела на Джон фон Нойман. Конвейерната обработка също попада в този клас архитектури, въпреки че имаме конкурентна обработка на различните етапи от различни инструкции в един момент. Конвейерната обработка не изпълнява едновременна обработка на различни инструкции, но подобрява производителността на процесора, от което се възползват почти всички процесори в днешно време. Техники, които използват възможност за паралелизъм при изпълняването на различни операции едновременно, също попадат в този клас архитектури. Тази форма на паралелизъм се нарича „паралелизъм на ниво инструкции“ ILP (Instruction-Level Parallelism). При тези техники различни инструкции се изпълняват едновременно в различни функционални устройства на процесора, а не в различни ядра или процесори. Такива архитектури са суперскаларните и VLIW (Very Long Instruction Word) архитектури.

При VLIW архитектурата имаме много дълъг регистър за инструкции. Компилаторът събира множество независими инструкции, които комплектува и зарежда едновременно в един регистър за изпълнение в един момент. Целта е хардуерът да бъде възможно най-лесен за изработка. Работата върху анализирането и организирането на различните инструкции в един регистър се измества върху компилатора. Компилаторът е отговорен за откриването на паралелизъм между различни инструкции и ги разпределя към различните функционални устройства в процесора.



Фиг. 1.16 Архитектурата VLIW

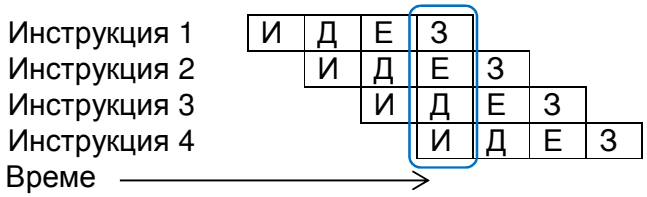
*ПД – Поток данни

Различните функционални устройства, като аритметично-логическо устройство и математически копроцесор, могат да работят паралелно. Инструкциите от регистъра се изпълняват едновременно в заключена стъпка. Недостатък при този тип архитектура е сложността на компилатора. Компилаторът трябва да открие независимите инструкции, които могат да се изпълнят паралелно, и да ги зареди в регистъра. В случай че не намери паралелизъм, отделни функционални елементи ще останат неизползвани. На неизползваните елементи се изпраща празна инструкция NOP (No Operation). Приложение, специално написано за едно поколение VLIW, се изпълнява незадоволително на друго поколение VLIW. Ако при едно поколение сме планирали няколко независими инструкции, то следващото поколение обикновено ще ни позволява много повече, което няма как да бъде предвидено. Благодарение на сложността на компилатора разработените подобрения са имплантирани при суперскаларните архитектури. Латентността при суперскаларните архитектури може да се толерира много по-добре с добавянето на подобренията, разработени при тези компилатори. И суперскаларните, и VLIW архитектурите планират различни операции да се изпълняват паралелно в различните ресурси на процесора.

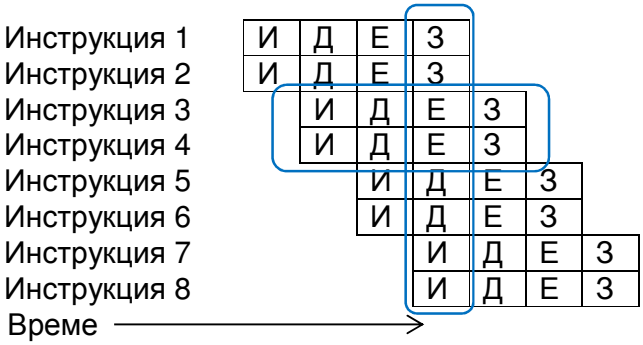
При суперскаларните архитектури анализът се прехвърля и изпълнява динамично от хардуера, за разлика от VLIW архитектурите, където анализът се изпълнява статично по време на компилацията. При суперскаларните машини прилагаме паралелизъм на ниво инструкции ILP по същия начин, без да нарушаваме последователността на инструкциите. По този начин в различните независими процесорни устройства се изпълняват различни инструкции по време на един машинен цикъл. Независими функционални устройства на процесора могат да бъдат няколко аритметични устройства, математически копроцесор или устройството за логическо изместване.

Въпреки че конвейерната архитектура и суперскаларните архитектури са различни технологии, суперскаларните процесори също са и с конвейерна обработка на инструкциите [14]. Ако различните етапи от изпълнението на една инструкция ги маркираме с: И – извличане, Д – декодиране, Е – изпълнение, З – запис, то обикновен пример за суперскаларен процесор ще изглежда по следния начин:

Конвейерно изпълнение на различните етапи от различни инструкции



Конвейерно суперскаларно изпълнение на различните етапи от различни инструкции



Фиг. 1.17 Конвейерно и конвейерно суперскаларно изпълнение

Недостатъците на този тип архитектура са усложненията при хардуера. Анализирането на потока от инструкции и техния паралелизъм се прехвърля изцяло към хардуера. На ниско машинно ниво е много трудно откриването на паралелизъм и зависимостите между различните инструкции. Това изисква допълнителни ресурси в процесора. Проверката на зависимостите между инструкциите отнема и от процесорното време.

Различните операции се изпълняват едновременно въз основа на анализ на зависимостите между операциите и операндите. Степента на вътрешен паралелизъм в един поток от инструкции е много ограничен. Следните инструкции са взаимно независими:

```
a = b + c
d = e + f
```

Ако имаме налични ресурси в един процесор, двете инструкции могат да бъдат изпълнени паралелно чрез суперскаларна или VLIW обработка.

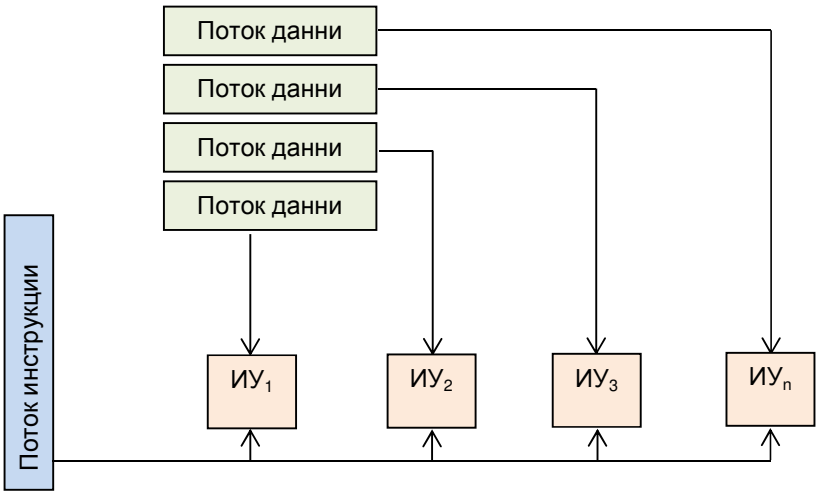
Противоположно на гореописаното, следните инструкции: са взаимно зависими и не могат да бъдат изпълнени паралелно:

```
a = b + c
d = a + f
```

Резултатът от първата инструкция е операнд във втората, което прави втората инструкция зависима и тя не може да бъде изпълнена паралелно, дори да имаме ресурси в процесора за едновременното им изпълнение. При добавянето на допълнителни функционални елементи в процесора времето за проверка на паралелизма нараства. В единия случай – времето за проверка при компилиране, в другия – времето на изчисление. Дори да имаме поток от инструкции, които са изцяло независими, те пак трябва да бъдат проверени. Ако след проверките се окаже, че имаме силна зависимост между инструкциите и нивото на паралелизъм е ниско, и при двата типа архитектури ще имаме реално забавяне при цялостното изпълнение на приложението.

Повечето компютърни системи работят в многопоточна среда. Множество приложения, включително и части от операционната система, работят едновременно. Когато имаме еднопроцесорна система, това се постига, като всички работещи приложения споделят процесорното време. Различните приложения могат да имат различен приоритет или всяка да взима равна част от процесорното време. Всяка програма или част от нея трябва да бъде заредена в паметта при нейното изпълнение. След приключването на нейната работа или полагащо се време програмата се изчиства и се зарежда следващата по ред. Постоянното четене от основната памет изисква много машинни цикли, както и достъп до шината за паметта. През времето за смяна на приложенията процесорът е в режим на изчакване. При суперскаларните и VLIW архитектури многото различни приложения, работещи конкурентно, ни дават възможност да подобрим производителността на системата. Ще постигнем това, като в конвейера зареждаме една инструкция за изпълнение от всяка програма. Например, ако имаме двадесет конкурентни програми, работещи едновременно, всяка двадесета инструкция в конвейера ще бъде от една програма. Този тип архитектура е известен като „многонишкова архитектура“. Голямото предимство при тези процесори е, че нямаме зависимост между последователните инструкции. Различните инструкции могат да бъдат изпълнени от различни функционални елементи на процесора едновременно, без да се прави проверка за зависимост между тях, тъй като те са от различни процеси. Недостатък на този тип архитектура е нуждата от много вътрешни регистри. За всяко работещо приложение трябва да имаме програмен брояч, съответния брой буфери, както и индекси, указващи коя инструкция на коя програма принадлежи.

1.3.2 SIMD multiprocessing



Фиг. 1.18 Единична инструкция оперира върху множество от данни

SIMD (Single Instruction Multiple Data) – единична инструкция оперира върху множество от данни. Това е клас многоядрени архитектури за паралелна обработка на данните, която използва постоянството при обработката на различни видове данни. Паралелизмът произлиза от изпълнението на еднаква операция върху различни данни. Този метод на паралелизъм има голямо предимство при обработка на масиви като видео или аудио обработка. SIMD може да работи по два различни начина: чрез обработка на данните като масив или с векторна обработка. Разликата между двата модела е в разпределението на задачите към различните ядра. Ако трябва да изпълним следните инструкции върху масив от данни:

```
LD    VR ← A[0:3]  – load – зареждаме четири променливи в регистрите
ADD   VR ← VR, 1   – add – добавяме едно към всяка променлива
MUL   VR ← VR, 2   – multiply – умножаваме резултата по две
ST    A[3:0] ← VR  – store – записваме резултата
```

Матрична обработка

P_1	P_2	P_3	P_4
LD A0	LD A1	LD A2	LD A3
ADD A0	ADD A1	ADD A2	ADD A3
MUL A0	MUL A1	MUL A2	MUL A3
ST A0	ST A1	ST A2	ST A3

Векторна обработка

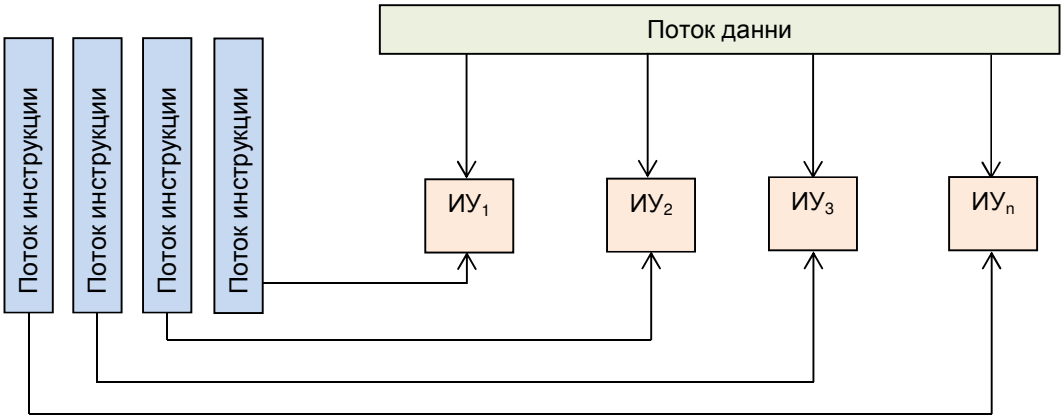
P_1 LD	P_2 ADD	P_3 MUL	P_4 ST
LD A0			
LD A1	ADD A0		
LD A2	ADD A1	MUL A0	
LD A3	ADD A2	MUL A1	ST A0
	ADD A3	MUL A2	ST A1
		MUL A3	ST A2
			ST A3

Фиг. 1.19 Матрично и векторно изпълнение

* VR – Масив с променливи за обработка

При матричната обработка всички ядра обработват една и съща инструкция в даден машинен цикъл. При векторната обработка едно ядро изпълнява конкретна инструкция от приложението. По този начин ядрата обработват последователно множеството от входящи данни в конвейерна форма, без да променят своята инструкция между различните цикли. Хардуерната изработка за векторна обработка на данните е видимо по-евтина. Отделните ядра не трябва да могат да поддържат различни инструкции. Допълнително предимство при векторната обработка е това, че данните са независими една от друга. Това от своя страна ни дава възможност за генериране на по-дълги конвейери. Имаме постоянен и предвидим достъп до паметта, което ни позволява да изтегляме данни от основната памет предварително. Недостатъците на SIMD архитектурите са няколко. Този тип обработка на данни е тясно специализиран и е подходящ само когато имаме постоянен паралелизъм. При задачи с широко приложение векторните компютри изпълняват приложенията изключително бавно. Изискването за големи регистри увеличава консумираната електроенергия и големината на чипа. Все още използването на алгоритми със SIMD инструкции изисква специфично указание от програмиста, като повечето компилатори не генерират SIMD инструкции автоматично. Също така много често шината за достъп до паметта се претоварва.

1.3.3 MISD multiprocessing



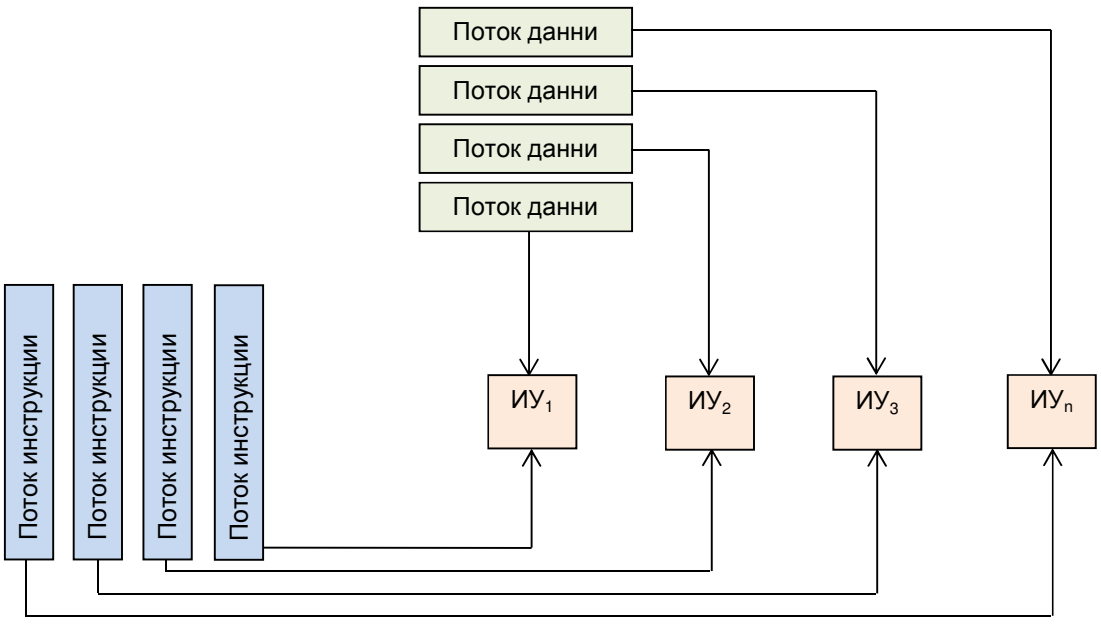
Фиг. 1.20 Различни инструкции оперират върху един елемент данни

MISD (Multiple Instruction Single Data) – множество инструкции оперират върху един елемент данни. При този тип архитектури паралелизъм се постига, като много процесорни елементи обработват едновременно един поток от данни. Интересът към MISD архитектурите е малък, тъй като практическите приложения за този тип задачи не са често срещани. Този тип архитектури може да се ползват при машини с ниска

толерантност към грешки. Един поток от данни се обработва от няколко процесорни елемента, обработените данни се сравняват и при откриване на грешка, тя се коригира спрямо принцип, който сме избрали, или се връща за повторна обработка. Такъв тип архитектури може да се срещне при приложения за космическите програми или високонадеждни машини. Други подходящи за този вид обработка приложения са за аудио и видео обработка. Когато множество честотни филтри обработват един поток от информация, можем да дадем задача на всеки процесорен елемент да изпълнява различни операции върху същия поток. Също така може да имаме необходимост от този тип архитектура за разсекретяването на криптографирани съобщения. Много различни алгоритми могат да работят върху едно кодирано съобщение.

Една от основните причини за ниския интерес към MISD е, че приложенията за този тип архитектури се срещат рядко и разработването им е ограничено. Друга причина е, че масово използваната архитектура MIMD, позволяваща обработването на множество данни с различни инструкции, която ще разгледаме по-късно, лесно може да се конвертира за ползване като MISD.

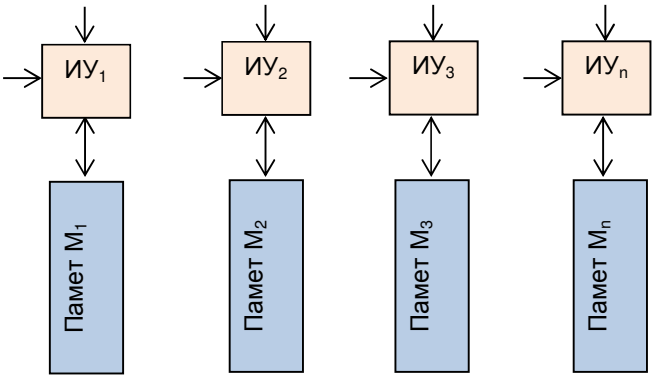
1.3.4 MIMD multiprocessing



Фиг. 1.21 Различни инструкции оперират върху множество от данни

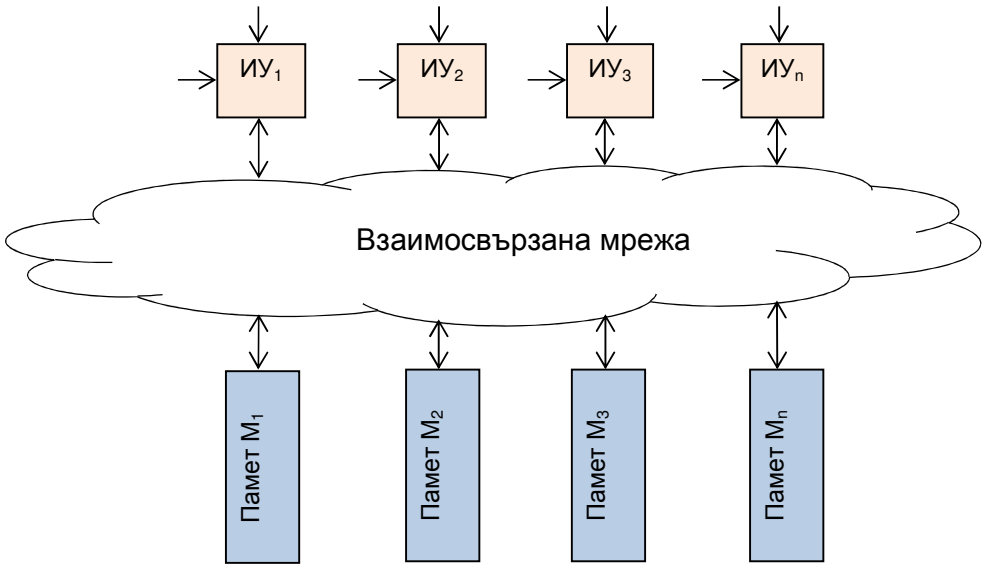
MIMD (Multiple Instructions Multiple Data) – множество инструкции оперират върху множество данни. Това са многопроцесорни, многоядрени или машини с многонишкови процесори. MIMD архитектурата е най-сложна като техническо изпълнение, но въпреки

това е много често срещана, защото позволява по-голяма гъвкавост, както и по-добри възможности за мащабно подобрение. Архитектури от типа MIMD трябва да могат да обработват няколко процеса асинхронно и независимо. Във всеки един момент различни изчислителни елементи трябва да изпълняват различни инструкции върху различни данни. Целта на този тип архитектура е постигането на паралелизъм при обработка на приложенията, с което ще подобрим скоростта на тяхното изпълнение. Освен скоростта на изпълнение, при много приложения или части от тези приложения, такива архитектури могат да постигнат по-ниска електрическа консумация. Ако разделим едно ядро на четири, работещи четири пъти по-бавно ядра, теоретично те ще изпълнят една задача за същото време. Консумираната енергия от един процесор е тясно свързана с честотата, с която работи. Намаляването на общата честота ще доведе до намаляване на консумацията за цялото изпълнение на задачите. Друга цел за разработването на многопроцесорни архитектури може да бъде и улесняване на изработката на един процесор. В зависимост от целите няколко прости ядра могат да изпълнят брой задачи значително по-бързо и по-ефективно от едно усложнено ядро, поддържащо голям брой инструкции. Така се постига не само по-лесното проектиране на процесора, но и по-евтината му изработка. Както при MISD, този тип архитектури може да се ползват при машини с ниска толерантност към грешки. Една задача може да се изпълни на няколко ядра едновременно, след което да се сравни резултатът. Друг вариант е, че при отпадане на едно от процесорните ядра ще можем да продължим работа с останалите. За да се възползваме от MIMD архитектурата, трябва да имаме възможност за паралелизъм на задачите. Това може да се постигне при паралелизъм на една задача, като я разделим на множество сегменти, които да обработваме едновременно. Друг вариант е като обработваме различни задачи едновременно, които нямат нищо общо една с друга. Всички тези фактори насочват разработването на многоядрени системи дори в мобилните устройства. MIMD архитектурите се делят на два подтипа: архитектури с разпределена памет и силно свързани архитектури с обща памет.



Фиг. 1.22 MIMD с разпределена памет

Архитектурите с разпределена памет не споделят глобална обща памет и общи елементи. Процесор от една система не може да адресира пространство от паметта на друга система. Това могат да бъдат многопроцесорни системи или дори многокомпютърни системи, свързани чрез обща мрежа. Всичките процесори комуникират чрез съобщения. Въпреки че имаме отделни елементи за различните ядра, отново се сблъскваме с проблем при синхронизацията. Ако имаме нужда от данни, ползвани от друг процесор, ние ще трябва да изчакаме тези данни да бъдат освободени и изпратени през комуникационната мрежа чрез пакети или съобщения. Свързването на всяка система с всяка друга няма да е икономически изгодно при голям брой системи. Тогава ще трябва да изберем различна топология на общата мрежа за сметка на ефективността на тази мрежа. Скоростта на предаването на съобщенията чрез общата мрежа е основният недостатък на архитектурите с разпределена памет.



Фиг. 1.23 MIMD с обща памет

При силно свързаните архитектури изчислителните ядра споделят общи елементи и памет. Програмният модел е подобен на еднопроцесорните многонишкови системи с тази разлика, че операциите върху общите елементи трябва да бъдат синхронизирани при самата обработка от всяко процесорно ядро. От гледна точка на програмиста този тип архитектури е по-лесен за разбиране, защото програмистът не трябва да се грижи за валидността на данните, заредени в различните части на паметта. При силно свързаните архитектури трябва да организираме и планираме изпълнението на различните задачи. Списък със сериозни проблеми при разработването на силно свързана архитектура трябва

да бъде предвиден и разгледан. Основният проблем е синхронизацията на споделената памет, поддържането на еднаквостта на всички копия на една променлива, ако са заредени в различни звена на паметта. Трябва да подсигурим подредбата на операциите, заредени в различните звена на паметта. Споделянето на общите ресурси и тяхното заемане от различните процесорни ядра не може да стане по едно и също време. Мрежата за комуникация между различните елементи, процесори или процесорни ядра също не може да бъде заета едновременно. Балансираното натоварване на отделните изчислителни звена трябва да бъде разгледано и планирано. Предимството при силно свързаните архитектури е комуникацията между различните процесорни ядра. Различните процесори могат да адресират споделени клетки от паметта и по този начин да обменят информация. Реално имаме много малко забавяне при обработка на данни от различни процесорни елементи. За този тип архитектура скоростта на паметта е критична, защото ще трябва да обслужва много изчислителни ресурси едновременно. Както при архитектурите с разпределена памет, взаимовръзките между различните процесорни елементи и на процесорните елементи с паметта могат да бъдат различни в зависимост от приложението.

Хибридни архитектури с различни нива кеш памет се възползват от предимствата на двата типа. Едно ниво кеш памет е разпределено за конкретно ядро, а друго ниво кеш памет е споделено между различните процесори. Може дори да внедрим междинно ниво, споделено само от двойка процесорни ядра.

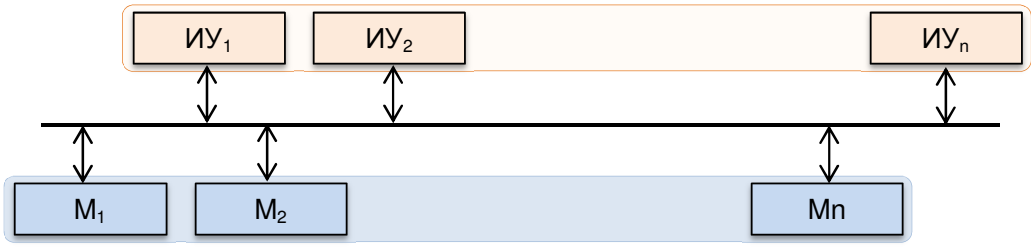
1.4 Компромиси при различните видове комуникационни мрежи

Независимо от това дали имаме многоядрена, многопроцесорна, или многокомпютърна система, начинът на свързване на различните звена ще е критичен за производителността на тази система. Връзките между компонентите може да са процесор с процесор, процесор с памет, памет с памет, както и входно-изходните устройства. Когато избираме типа на взаимовръзките между изчислителните ядра и паметта, трябва да вземем предвид цената на изработка и латентността на мрежата. Имаме още важни фактори, като консумацията на електроенергия, пропускателната способност, как ще работи с наличните ни процесори и още много други. Основите разлики между комуникационните мрежи имат няколко компонента:

- Топологията на връзките – определя метода, по който отделните компоненти са свързани. От това ще зависи по какъв маршрут ще преминават данните, надеждността на мрежата, латентността, пропускателната способност и сложността на изпълнение.
- Алгоритмите за комуникация – зависи как едно съобщение ще стигне от източника до приемника. Това може да стане по статичен или адаптивно променян маршрут.

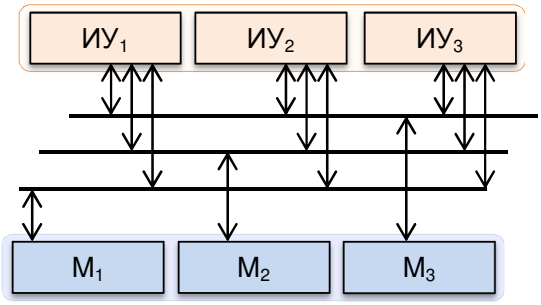
- Начинът за буфериране на съобщения и потока на данни – какво ще бъде записвано в самата мрежа, както и как фрагментираме и изпращаме пакетите; как ще ограничим подаваните данни, ако мрежата е пренатоварена.

При топология от типа обща шина всички звена са свързани към една шина. Това прави модела най-лесен за изработка и евтин за осъществяване, когато имаме малък брой елементи, прикачени към шината.

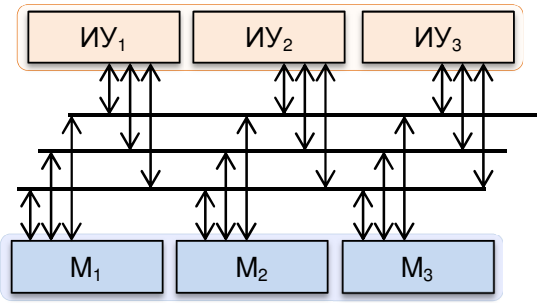


Фиг. 1.24 Топология обща шина с обща памет

Недостатък на този модел е общият ресурс, ползван от всички елементи. При голям брой изчислителни звена или клетки памет, прикачени към общата шина, мрежата бързо се претоварва и става неефективна. За подобряване на потока от данни с обща шина са разработени различни модели с поставянето на допълнителни шини, които позволяват паралелна комуникация между няколко елемента.



Фиг. 1.25 Топология MBSBMC

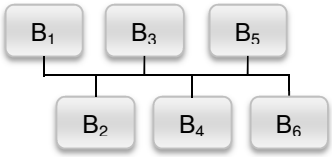


Фиг. 1.26 Топология MBFBMC

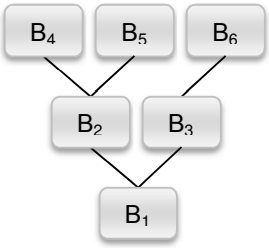
Като пример може да разгледаме многошинова многопроцесорна система, използваща няколко шини за комуникация паралелно. При тези разновидности отново можем да направим избор спрямо цената и резултата, който искаме да постигнем в зависимост от приложението. В примерите от Фиг. 1.25 и Фиг. 1.26 архитектурите от типа MBSBMC (multiple bus with single bus-memory connection) и MBFBMC (multiple bus with full bus-memory connection) позволяват различни процесори или клетки памет да комуникират по едно и също време. Това ще подобри обмена на информация, но ще оскъпи производството на мрежата, както и ще усложни дизайна на системата.

Една комуникационна мрежа може да бъде много различна в зависимост от приложението, за което ще я ползваме. При системи с разпределена памет или за комуникация между отделни възли може да имаме следните топологии:

Фиг. 1.27 Разпределена шина

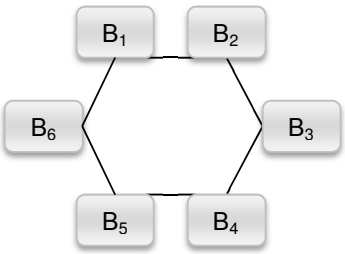


Фиг. 1.28 Обща шина

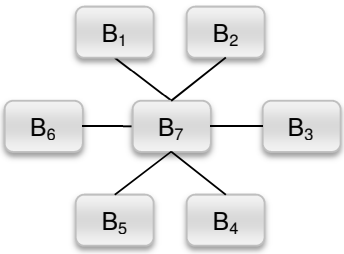


Фиг. 1.29 Дърво

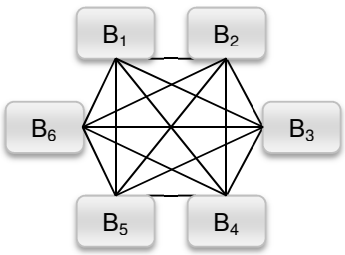
Предимствата на топология тип дърво са при специализиран тип задачи, изискващи локален обмен на данни – например на Фиг. 1.29 между възлите B_4, B_5 и B_2. Те са евтини и лесни за изработка. При задачи, изискващи комуникация между всички възли, коренът на дървото (B_1) се претоварва и забавя цялата система.



Фиг. 1.30 Пръстен



Фиг. 1.31 Звезда

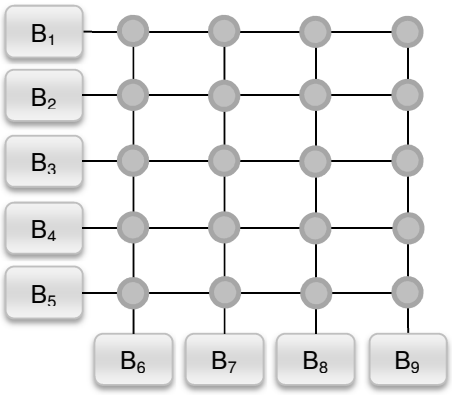


Фиг. 1.32 Напълно свързана

При топология от тип пръстен (Фиг. 1.30) предимствата са цената и неособената сложност на изработка. Самите връзки между възлите са малки и малко на брой. Големият недостатък е пропускателната способност на мрежата при по-голям брой звена. За подобряването на пропускателната способност имаме модели с двупосочна магистрала в пръстена, както и йерархична пръстенова топология. Добавянето на нови възли е изключително лесно, но с всеки добавен възел пропускателната способност на мрежата намалява. Топология йерархичен пръстен се получава, когато свържем няколко пръстена с друг или други пръстени. Добавянето на нови пръстени към основния или нови звена към по-малките пръстени е лесно мащабируемо. Поради ниската цена на изпълнение тази топология е предпочитана и често срещана в индустрията. Напълно свързаната мрежа от Фиг. 1.32 е най-бързият вариант и при него ще имаме най-малко забавяне между съобщенията. Това е възможно най-скъпата архитектура, която може да изберем. Освен

цената на производство, друг недостатък на този тип мрежа са усложненията при поставянето на всеки нов възел. По тази причина напълно свързаната мрежа не е подходяща при системи с по-голям брой възли.

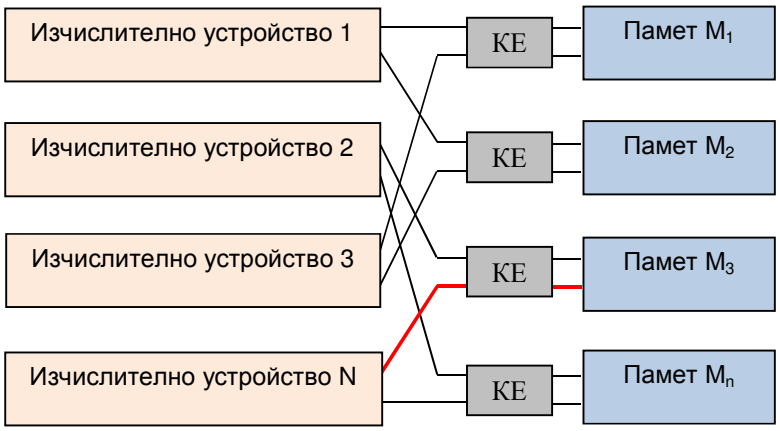
За постигане на задоволителна ефективност, надеждност на мрежата, както и за да запазим възможността всеки възел да комуникира с всеки друг, са разработени динамични мрежи. Динамичните мрежи се изпълняват чрез комуникативни звена, които превключват към други звена или възли в зависимост от заявката. При тази архитектура превключването може да бъде изпълнено апаратно или чрез инструкция в предавания пакет. При апаратно превключване активираме комуникативните звена в конфигурацията, която ни е необходима преди комуникацията, след което изпращаме съобщението между възлите. При комуникация чрез пакети не е необходимо да настройваме възлите, а изпращаме пакета, след което комуникативното звено декодира пакета и го препраща спрямо инструкцията в самия пакет. Пример на динамична топология е матричен модел мрежа – Фиг. 1.33.



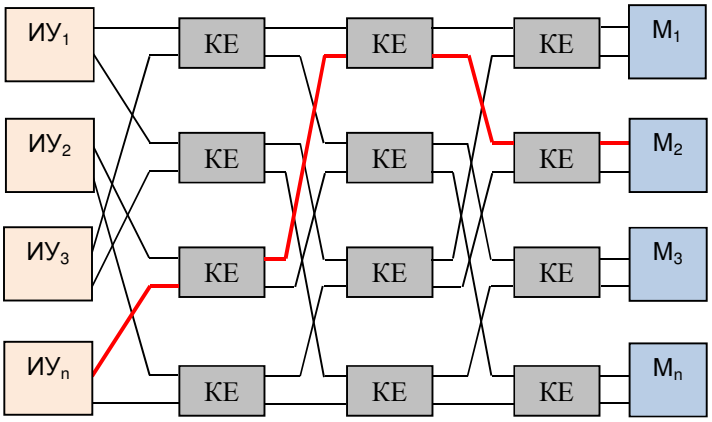
При топология от тип матрица имаме взаимовръзка между всички възли, но само един възел може да ползва споделена връзка в определен момент. Това позволява едновременна комуникация между възли, които не споделят обща шина.

Фиг. 1.33 Матрица

Недостатък на матричната топология е цената за изработка, както и трудността при добавяне на голям брой възли. За подобряване на матричната топология са разработени модификации с поставянето на буфери между елементите или при комутационните звена, с които да се подобри пропускателната способност на мрежата. С поставянето на буфери сложността на мрежата се увеличава и оскъпява допълнително. Стремежът за изпълнение на по-евтина топология от типа матрица и все пак с по-добър поток на данните, сравнено с типа обща шина, довежда до разработването на многостъпална топология.



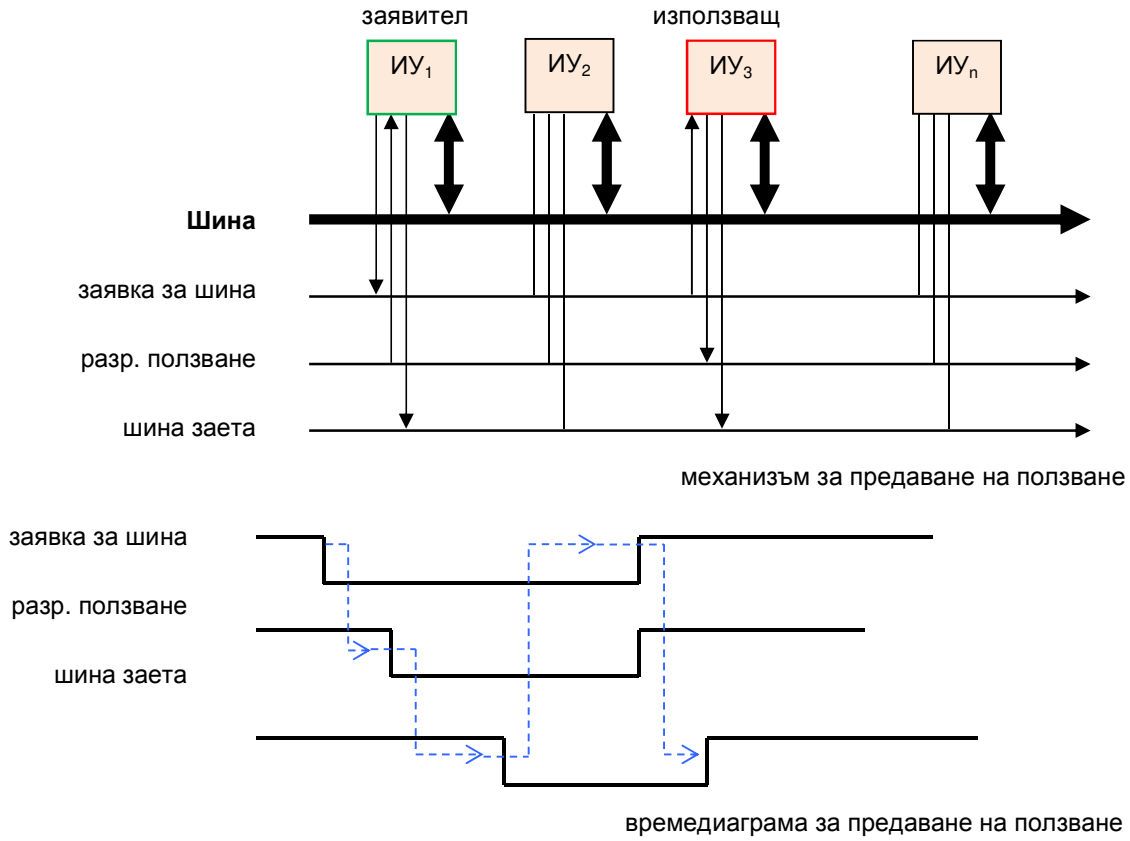
Фиг. 1.34 Комутационна мрежа



Фиг. 1.35 Многостъпална комутационна мрежа
*КЕ – Комутаторен елемент

Това са индиректни мрежи, при които комуникацията между различните възли се постига чрез каскада от комутатори. Комутаторите позволяват двупосочна комуникация. Многостъпалните мрежи могат да бъдат с активни превключващи елементи или превключването да се извършва с пакети. При активните елементи комутаторите се превключват преди комуникацията между две звена. Веднъж установен пътят, двата комуникиращи елемента започват комуникация. Недостатък на този метод е, че никой друг не може да ползва заетите комутатори. Другият вариант за ползване на многостъпална мрежа е чрез пакети. Самият пакет съдържа информация с адреса на получателя. След достигане на комутаторен елемент от мрежата, в зависимост от адреса в пакета комутаторният елемент динамично го предава към подходящия изход. По този начин можем да изпращаме пакети от различни елементи, без да сме блокирали целия път от едно звено до друго. Потенциално може да загубим време за декодиране на адреса, както и да усложним мрежата. Ако сме добре запознати с топологията на компютърната архитектура, може да поставим комуникиращите изпълнявани приложения близо едно до друго, което ще доведе до сериозно скъсяване на пътя за комуникация.

При всички видове топологии, с изключение на напълно свързаната (Фиг. 1.32), имаме проблем, когато две различни звена се опитват да ползват общ ресурс. Имаме три метода за разрешаване на такъв сценарий: като откажем една от комуникациите, чрез буфериране или като отклоним пътя на комуникацията. При отклоняване ползваме път, който е свободен, въпреки че това не е най-краткият път между звената. Така пакетът може да стигне до получателя през по-дълъг маршрут, но няма да бъде отказан или буфериран. Този метод ни позволява да пращаме пакет всеки път, когато имаме свободен маршрут, дори това да не е предвиденият и най-кратък маршрут. Също така ни позволява да разработим комуникационна мрежа без буфериращи елементи. Сериозен недостатък на тази мрежа е при претоварване на мрежата. Веднъж натоварена мрежата, препратените пакети допълнително натоварват звената, като преминават през твърде дълги маршрути. Това може да доведе дори до безизходен застой на мрежата. При буферирането комуникационните елементи записват предаваните данни и ги препращат чак когато се освободи мрежата. Спрямо времето за ползване на мрежата системата може да бъде синхронна или асинхронна. При синхронната система времето за комуникация между два възела е предварително известно. Асинхронните мрежи позволяват възлите да избират използваното време на шината – Фиг. 1.36.



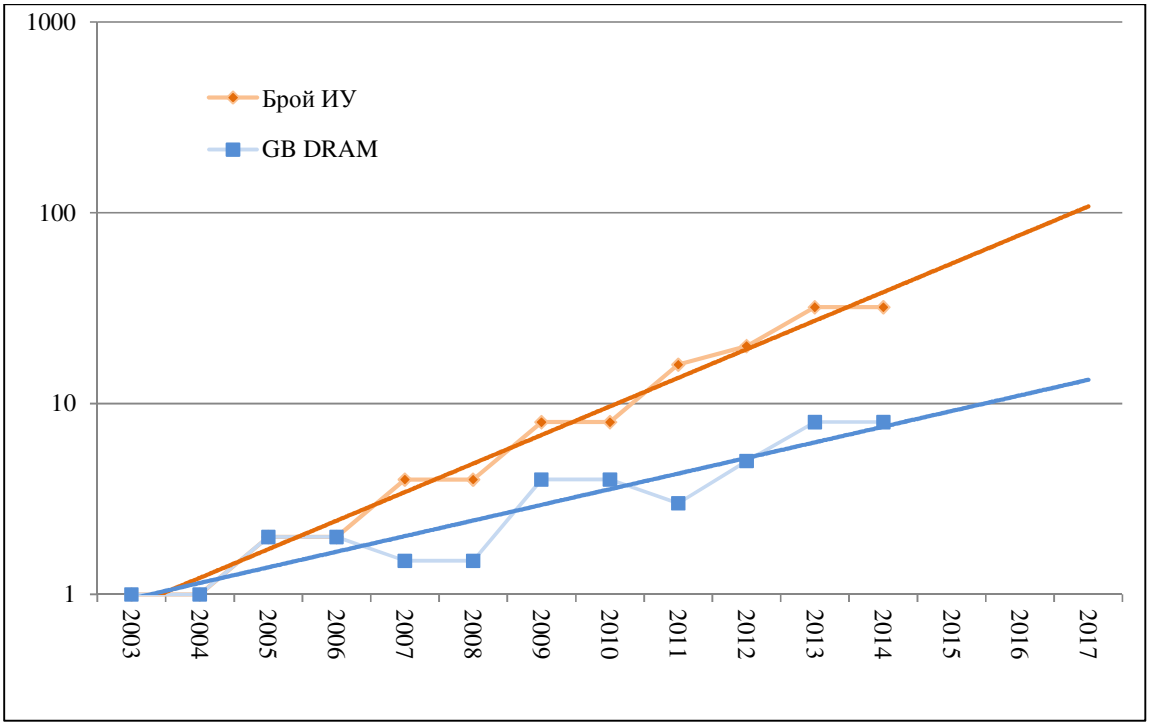
Фиг. 1.36 Блоксхема и времедиаграма за предоставяне на право на ползване

Когато разглеждаме многопроцесорни системи или многоядрени системи, трябва да обърнем внимание на комуникацията между различните възли в самата система. Мрежите за комуникация между ядрата в многопроцесорна система се превръщат в актуален проблем през последното десетилетие. Тези мрежи се класифицират като мрежи върху чип. Ползваме ги, за да свързваме ядра, звена на кеш паметта, контролерите и останалите елементи от един или няколко процесора. Като цяло мрежата върху чип е заета предимно със заявки от кеш паметта или със съгласуване на променливите в тези кеш звена. Недостатъците на различните видове топологии, които разгледахме до момента, са валидни и за мрежите върху чип. Това забавяне на комуникацията между различните звена в процесорите задълбочава проблема с комуникацията между процесора и основната памет.

1.5 Изводи от проучването

Разработките върху компютърните архитектури в днешно време са съсредоточени основно върху производителността и консумираната електроенергия. Повечето изследвания са съсредоточени върху прилагането на паралелизъм, свързан с обработка на няколко инструкции в един цикъл. Други изследвания са насочени към конкретни проблеми и архитектурите са строго специализирани изцяло само за задачата, която трябва да бъде изпълнена. Като цяло може да се каже, че при универсалните компютърни системи разработките са в областите на: усъвършенстване на интерфейса между процесора и паметта, вътрешният дизайн на процесорите чрез увеличаване броя на ядрата, йерархия на паметта, както и многопроцесорни системи. Въпреки всички разработки, проучвания и голямо разнообразие, най-критичното място, засягащо бързодействието на компютрите, остава комуникационната мрежа между основната памет и процесора.

Главните тенденции, които засягат основната памет, са нуждата от обем, бързият достъп, консумираната енергия и бъдещото развитие на технологиите. Обемът и времето за достъп се влияят от разработките върху мултипроцесорните системи, при които общата памет трябва да обслужва всички ядра. Съвместното сътрудничество между HP Labs, Мичиганския университет и AMD относно тенденциите при развитието на паметта и многопроцесорните архитектури показва, че има разминаване в развитието на плътността на паметта спрямо ядрата [7] [86]. Броят ядра се удвоява приблизително на всеки две години, докато капацитетът на паметта се увеличава на всеки три години. Капацитетът на паметта към процесорните ядра ще намалява с 30% на всеки две години.



Фиг. 1.37 Капацитетът на паметта към процесорните ядра

Тенденциите по отношение на пропускателната способност на шината на паметта са още по-тревожни. Пропускателната способност на шината се увеличава приблизително с 10% всяка година. Самите приложения с времето имат все по-голяма нужда от достъп до данни. Програмистите добавят нови функции към приложенията за сметка на паметта. При многопроцесорните системи можем да изпълняваме много независими приложения върху независими процесорни ядра, което ще задълбочи допълнително проблема с ограниченията върху паметта.

За облекчаване на достъпа до оперативната памет почти всички съвременни процесори са разработени с различни нива кеш памет, както и регистри, работещи със скоростта на ядрото. Пропуск на прочитане от кеш паметта или провален опит поради липсващи данни от локалната памет на процесора означава заявка за достъп до основната памет, който е много по-бавен.

По последни данни на Intel приблизителното време за достъп до различните кеш нива на процесор Intel Core E7 Xeon е:

1 цикъл за четене от регистър;
4 цикъла за четене от L1 кеш;
10 цикъла за четене от L2 кеш;
40 цикъла за четене от локален L3 кеш;
75 цикъла за четене от споделен L3 кеш;
и няколкостотин цикъла за четене от основната памет.

Съществуват три вида пропуски при достъп до локалната памет на процесора: пропуск при четене на инструкции, пропуск при четене на данни, пропуск при запис на данни. Пропускът при четене на инструкции предизвиква най-голямо забавяне. Процесорът трябва да преустанови изпълнението на приложението до извличането на инструкцията от основната памет. Пропускът при четене на данни предизвиква по-малко забавяне, защото процесорът може да продължи изпълнението на независими инструкции през времето на изчакване на данните. След като данните са получени и обработени, зависимите инструкции също се изпълняват. Пропускът при запис предизвиква най-малкото забавяне, защото данните могат да бъдат временно съхранени, докато ядрото продължи със следващата инструкция. Огромна част от разработките са насочени главно към анализ на поведението на кеш паметта в опит да се намери най-добрата комбинация от размер на кеша, асоциативност, размер на блоковете и взаимовръзките в кеша. Самият анализ на пропуските от кеша е труден, като изисква програми за симулация. Последователни симулации върху различни кеш модели разглеждат ефекта на всеки вариант върху общия брой пропуски. Опитите да се отчетат разликите от всички променливи по време на симулация, налага разделянето на пропуските на три основни типа: студени пропуски, пропуски заради размера и конфликтни пропуски. Студените пропуски са пропуски, породени при първото извикване на адрес в паметта. Предварително извличане и зареждане на данните в кеша може да помогне в този случай. Пропуските, породени от размера на паметта, се случват независимо от асоциативността, а само и изцяло от крайния размер на кеша. Графиката на пропуските към размера на паметта дава известна степен на зависимост, която може да измерим. Важно е да се отбележи, че заетостта на кеша няма значение за тези измервания, тъй като почти през цялото време целият кеш на процесора е запълнен с различни блокове от основната памет. Конфликтните пропуски се дължат на определеното ниво на асоциативност, както и на метода на заместване, по който избираме кои вече заредени данни да бъдат заменени за сметка на новоизвиканите. Поставянето на изчислителни ресурси в основната памет ни предоставя допълнителни възможности за избягване на пропуски при четене на кеша. Текущите разработки са фокусирани върху конфликтните пропуски. Възможностите за предварително зареждане на блокове от основната памет в конвенционалните процесори не са много и включват допълнителни инструкции, зададени в кода от програмиста. Допълнителен процесорен елемент в паметта ще ни позволи паралелно работещ процес спекулативно да зарежда блокове от основната памет в кеша, които смята, че ще бъдат ползвани непосредствено. Блокът от основната памет може да бъде както с инструкции, така и с данни. Предварителното успешно презареждане на инструкции ще има най-голям

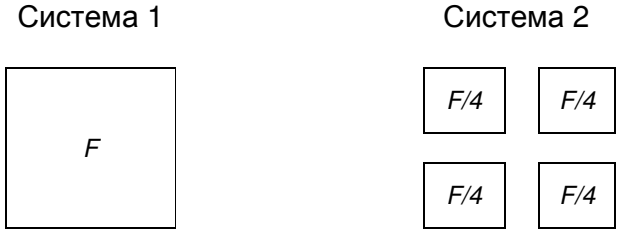
ефект върху общото време за обработка на едно приложение, тъй като пропускът при четене на инструкция забавя системата значително.

Поставянето на изчислителни ресурси в паметта ще доведе до осезателно намаляване на електрическата консумация. Добавянето на допълнителна оперативна памет за подобряване на производителността не винаги е подходящо решение, особено когато имаме ограничение върху консумираната електроенергия. DRAM чиповете консумират електроенергия, дори когато не се използват, поради периодичното опресняване на клетките. Към консумацията от паметта трябва да добавим консумираната електроенергия от процесора. Няколко фактора допринасят за общата консумация на едно изчислително ядро: динамичната консумация, консумацията на логичните елементи и загубите при оттечки на транзисторите. Динамичната консумация на електроенергия е основен фактор и тя е пропорционална на честотата, с която работи процесорът.

$$Pdyn=FV^2C$$

Pdyn – Dynamic Power – Динамична консумация
F – Frequency – Честота
V – Voltage – Волтаж
C – Capacitance – Капацитивно съпротивление

Като пример: ако разпаралелим едно приложение в четири ядра, можем да намалим честотата на тези ядра на четири и ще изпълним приложението за същото време, както с едно бързо ядро.



Фиг. 1.38 Сравнителна схема на различен брой ядра с еднакъв брой елементи

Ще намалим общата консумация за изпълнение на тази задача без загуба на време и производителност. При изпълнението на приложение от изчислителните ядра в паметта отпада нуждата от комуникация между основния процесор и паметта, което ще доведе до допълнително намаляване на консумираната енергия от шината до паметта.

Обработването на данни директно върху паметта ще има голямо предимство и при приложения, изискващи обработка върху голям обем от данни. Такъв пример може да бъде търсенето от база данни. Няма да има нужда цялата база от данни да премине през процесора и шината на паметта, когато може търсенето да се извърши директно върху основната памет.

Цел и задачи на дисертацията

Целта на дисертацията е да се предложи модел на компютърна архитектура, използваща допълнителни изчислителни елементи в основната памет. Изчислителните ресурси в паметта предлагат спектър от възможни подобрения при изпълнение на изчислителния процес. Подобренията могат да бъдат осъществени както съвместно от софтуера и хардуера, така и изцяло от хардуера. Подобренията, позволяващи изцяло хардуерно изпълнение, ще останат скрити за програмиста и компилатора. Това ще позволи внедряването и поддържането на съществуващи приложения и операционни системи без допълнителни изменения.

Скоростта на процесорите и размерът на паметта се увеличава много по-бързо от възможностите на шината за пренос на данни. Проведени тестове показват, че на всеки четири такта изчислителното устройство губи три такта в изчакване на данни от паметта. С увеличаването на скоростта на изчислителните устройства и размера на паметта с всяко ново поколение сериозността на проблема се задълбочава.

Трябва да отбележим, че дори в най-добрия случай времето за достъп до кеш паметта не е един процесорен цикъл. При съвременните процесори първото ниво кеш памет работи с латентност четири цикъла. В най-добрия случай, за достъп до данни от паметта изчислителното устройство ще трябва да изчака четири машинни цикъла. В най-лошия случай, при липса на търсените данни в кеш паметта може да се стигне до петстотин и повече машинни цикъла за извличане на търсените данни от вторичната памет.

Основните идеи за използване на изчислителните ресурси в паметта са за обработка на независими приложения. Също така е възможна обработката чрез ресурсите в паметта на различни сегменти от едно приложение и изпращането на критичните секции към основния конвенционален процесор. Друга възможност е обработката на различни сегменти от едно приложение чрез ресурсите в паметта и изпращането на секциите, изискващи повече ресурси за обработка, към основния конвенционален процесор. Допълнителните изчислителни ресурси в паметта разгръщат възможности за разработки в различни сфери:

- компресиране на потока от данни по общата шина;
- трансформация на комуникацията по шината от ниво сигнали в ниво пакети;
- изпълнение на помощни потоци за предвиждане на разклонения;
- изпълнение на помощни потоци за презареждане на данни от системната памет.

В дисертационния труд са разгледани подробно някои от споменатите по-горе подобрения, като са симулирани и емулирани техните изпълнения. Резултатите от

тестовете са сравнени със съществуващи конвенционални системи за определяне на ускорението след конкретно подобрение.

Основните задачи на дисертацията са:

- Проучване на текущите разработки и съществуващи предложения по архитектури с изчислителни елементи в паметта;
- Предложение за многопроцесорна архитектура с изчислителни елементи в паметта и създаване на аналитичен модел, описващ основните характеристики;
- Разпределение на различни изчислителни функции и операции между процесорните елементи и основния процесор с промяна на основната диаграма за изпълнение на инструкциите в съответствие с предложената структура и разделяне на изчисленията;
- Провеждане на симулационни изследвания на конвенционални компютри и на модели на компютри с изчислителни ресурси в паметта за сравнение на общата производителност на различните архитектури и намаляване на информационния поток между централния процесор и паметта;
- Проектиране на основните възли и елементи от предложените структури с използване на съвременни технологии за хардуерно проектиране.

Разгледани са практическата приложимост и полезност на материалите от дисертационния труд. Темата е актуална и дисертационният труд има висока научна и приложна стойност. Добавянето на нови елементи към микроархитектурата, без съществени промени в архитектурата на системните инструкции, ще позволи да ползваме текущите разработки в тази област, като новите предложения останат прозрачни за софтуера от гледна точка на програмиста.

ГЛАВА II. Архитектура, базирана на изчислителни звена в паметта

Много често срещана грешка при първоначалния дизайн на многопроцесорна или многоядрена архитектура е очакването, че с добавяне на повече процесори или процесорни ядра в процесора ще намалим пропорционално времето за обработка на приложението. В действителност няколко основни недостатъка на многопроцесорните и многоядрените системи не позволяват това. Времето за изпълнение на едно приложение е ограничено от изпълнението на най-бавния сегментиран елемент на това приложение, ако то изобщо може да бъде сегментирано, от достъпа до споделени ресурси между различните сегменти на едно приложение или различни приложения, както и от дисбалансираното сегментиране на програмата. Голяма част от проучванията и разработките при многопроцесорните архитектури са за преодоляване на тези проблеми.

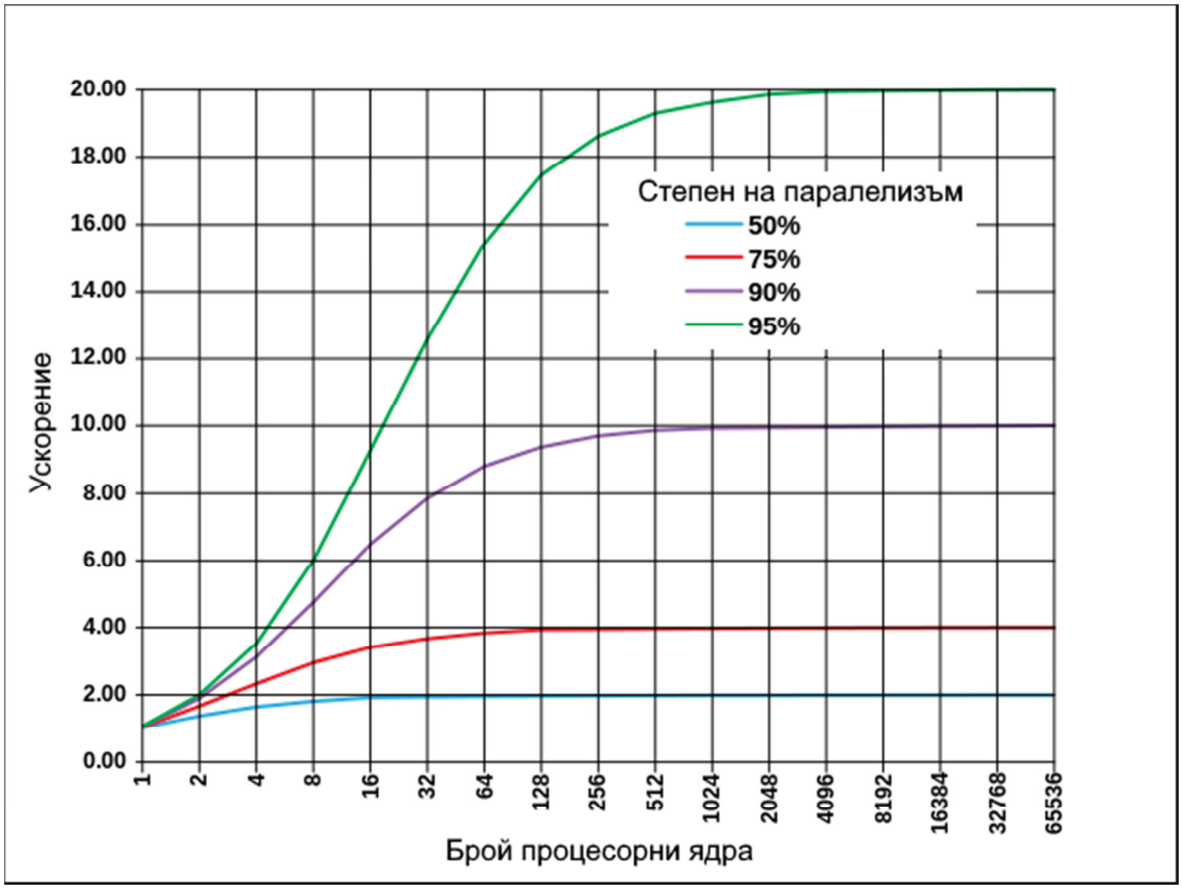
2.1 Текущи ограничения и недостатъци на многопроцесорните системи

Законът на Амдал илюстрира ограничението на ръста на производителността при увеличаване на броя изчислителни звена. Програмите може да разделим на два типа: програми, които могат да бъдат изпълнени само по стандартния последователен начин, и програми, които можем да изпълняваме паралелно. При паралелна обработка няколко сегмента от едно и също приложение могат да бъдат изпълнени по едно и също време. Програмите, позволяващи паралелна обработка, са най-ефективни при многопроцесорни архитектури, но въпреки това и те имат своите ограничения. Времето за цялостната обработка на приложението зависи от времето, необходимо за обработка на един от сегментите на това приложение, както и от броя ядра, с които разполагаме:

$$Sp=1/((1-P) + P/N)$$

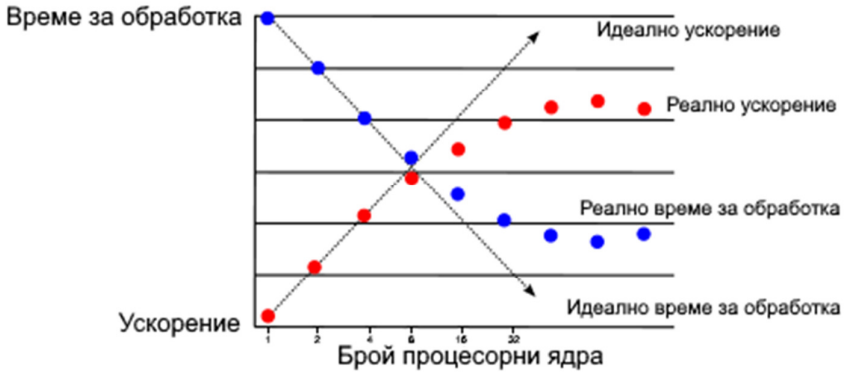
Sp – Ускорение на системата – първоначална скорост към подобрената скорост
P – Броя на конкурентните сегменти, които ни позволява приложението
N – Броя на процесорните ядра

Тази формула ни показва какво ще стане, ако увеличаваме процесорните ядра. Вижда се, че дори да увеличим ядрата до безкрайност, времето за изпълнение на програмата ще зависи изцяло от най-бавния сегмент на приложението (Фиг. 2.1). Така винаги достигаме до момент, при който, дори да добавим допълнителни процесорни ядра, няма да подобрим времето за изпълнение на приложението.



Фиг. 2.1 Ускорение на системата спрямо броя ядра и степента на паралелизъм

Друго ограничение при паралелна обработка на едно приложение е синхронизацията на операциите. Операции, обработващи общи променливи, не могат да бъдат паралелизирани. Когато един сегмент от приложението работи с променлива, необходима на друг сегмент от приложението, то вторият сегмент трябва да изчака освобождаването на тази променлива. Сегментите може да ползват както общи променливи, така и общи ресурси. По същия начин, както при променливите, различните сегменти трябва да изчакат общият ресурс да бъде освободен, преди да могат да го ползват. Това изчакване води до забавяне на изпълнението на цялото приложение.



Фиг. 2.2 Паралелно забавяне

Комуникацията между различните сегменти води до допълнително забавяне. Не всеки път при сегментирането на едно приложение подобряваме времето за изпълнение. Както отбелязахме по-горе, сегментите може да имат нужда от общи променливи или ресурси, необходими на конкурентни сегменти. Така те не само трябва да изчакат освобождаването на тези променливи, но трябва да комуникират помежду си. Времето за комуникация задълбочава проблема с изчакването. Така при прекаленото сегментиране на едно приложение може да увеличим времето за неговото изпълнение.

Неравномерното сегментиране на приложението и разпределението на задачите върху различните ядра е друг проблем, за който ще трябва да се грижим при многопроцесорните системи. Паралелните сегменти могат да имат различна сложност и време за изпълнение. Това може да се дължи на лошо сегментиране на приложението или на недобро зареждане на нужните данни в кеш паметта за някои от сегментите. Този проблем ще доведе до забавяне на паралелната част от приложението.

2.2 Съгласуваност на данните от кеш паметта

Различните нива кеш памет, както и разпределените блокове от едно ниво, принадлежащи към различни ядра или процесори, могат да съдържат копия от една и съща променлива. Когато стойността на тази променлива е променена, новата стойност трябва да бъде обновена във всичките ѝ копия. Съгласуването на променливите в различните кеш нива, както и на променливите в разпределената памет при системите е един от основните проблеми на съвременните компютри, независимо дали са многоядрени, или не. За съгласуваност на кеш паметта е необходимо да гарантираме еднаквост на променливите, заредени във всички кеш звена и паметта. Трябва да гарантираме коректността на променливите след тяхното обновяване от различните изчислителни устройства. Методите за съгласуваност трябва да подсигурят разпространението на обновената стойност, както и последователността в глобалната подредба за всички ядра.

За съгласуване на паметта, след като разберем, че една променлива е обновена, имаме два основни начина за обновяване на всички копия на тази променлива. Обновяването на модифицираните променливи може да става веднага след запис или като ги маркираме като невалидни. Ако сме избрали модел, при който маркираме блоковете от паметта, съдържащи модифицирана променлива като невалидна, тогава заедно със сигнала за запис разпространяваме сигнал за маркиране на всички блокове, съдържащи копия на променливата като невалидни. При необходимост процесорното ядро ще извлече

блока отново от паметта. Алтернативен метод е веднага след модификация да обновим всички блокове, съдържащи копия на променливата. Заедно със сигнала за запис едновременно разпространяваме новата стойност до всички копия. За сравняване на двата метода на обновяване на променливите трябва да знаем колко често ще ни се налага да записваме нова стойност на променливата, както и колко процесорни звена могат да ползват една и съща променлива в това време. Ако имаме разредено обновяване на споделени променливи, методът с автоматичното обновяване на всички копия ще бъде подходящ, защото ще си спестим времето и трафика по инвалидиране на променливите и следващото обновяване. Ако обаче едно ядро прави чести промени по споделена променлива, без другите ядра да я ползват, трафикът по обновяването на променливата ще бъде излишен. Тогава методът с маркиране на блока като невалиден ще бъде по-подходящ. Само ядрата, ползващи тази променлива, ще дадат нова заявка за обновената стойност. Недостатъкът на този метод се проявява при чести записи от различни ядра. Когато две или повече ядра започнат да правят промени по една и съща променлива, се появява проблем, наречен пинг-понг (многократно инвалидиране на една променлива от различни ядра).

2.2.1 Програмно съгласуване на променливите в различните кеш звена

Всеки път, когато говорим за кеш съгласуваност, ние косвено говорим за автоматичен хардуерен контрол на променливите. При проектиране на компютърна система ние имаме опция и за софтуерно съгласуване. Една от възможностите е да съгласуваме променливите чрез виртуалната памет. На всяка страница от виртуалната памет може да поставим допълнителни битове, указващи дали страницата е споделена и дали информацията в нея е актуална. Когато направим запис в споделена страница от виртуалната памет, ще изчистим всички останали копия.

Друг вариант за софтуерно съгласуване на променливите е да подсигурим инструкции за изчистване на блок от паметта чрез системни инструкции. Чрез архитектурни инструкции ще задаваме команда за изчистване на: блок от процесорната локална кеш памет, инструкция за изчистване на блок от локалния кеш на всички останали процесори или процесорни ядра, инструкция за изчистване на всички блокове, съдържащи се в определено кеш звено. Разбира се, изчистването на всички копия на определен блок ще заеме твърде много ресурси и никога не е подходящ вариант.

Предимството на този тип съгласуване ще е лесната изработка на компютърната микроархитектура. Софтуерното съгласуване е подходящо за устройства, които нямат голяма нужда от съгласуваност. Ако програмистът е експерт и работи с малко данни,

които имат нужда да бъдат споделяни между различните ядра, той може да избира кои данни да седят по-близо до ядрото. Това може да подобри производителността на системата. С други думи, когато няма много променливи, които се ползват от различни процеси в един и същи момент и при паралелни архитектури, изпълняващи независими процеси, ще имаме полза от този тип поддържане.

Недостатък за създаването на програма, която да се грижи и за съгласуването на паметта, е допълнителната работа за изработката на програмата. Самото съгласуване е изключително трудоемко и създава предпоставки за допускането на грешки.

2.2.2 Апаратно съгласуване на променливите в различните кеш звена

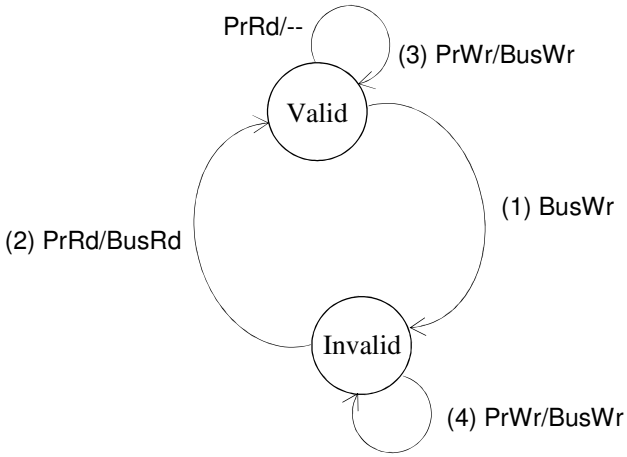
При апаратното автоматично съгласуване хардуерът управлява движението на данни между различните нива на паметта. Програмистът няма нужда да се грижи за копията на променливите, както и да е запознат с микроархитектурата на компютърната система. Апаратното организиране улеснява работата на програмиста, но изпълнението на приложенията не е оптимизирано и усложнява изработката на микроархитектурата.

Основно два механизма се ползват за разпознаване на обновена променлива:

- Чрез указател – логически централизиран общ указател следи всички блокове от паметта и в него се записват техните състояния. Указателят регистрира кой блок е зареден в различните звена памет и координира заявките за промени. Когато едно процесорно звено даде заявка за обновяване на променлива, указателят прави запис на тази заявка, като записва, че този блок принадлежи на процесорното звено. Указателят следи кое процесорно звено е дало заявка за четене или обновяване на определен блок и неговите копия. Указателят координира кой блок трябва да бъде маркиран като невалиден или обновен. Указателят е логически централизиран, не е необходимо да имаме обща шина или общ ресурс, което прави този метод лесен за изпълнение при голям брой процесорни звена. Недостатъкът е забавянето от допълнителната стъпка за обръщението към указателя: първо процесът трябва да се обърне към указателя, да провери състоянието на копията на заявения блок от паметта и чак тогава да отговори на запитването. Друг недостатък на този метод е изискването за допълнително пространство за съдържанието на указателя, в което можем да записваме състоянията на всички блокове от паметта.
- Чрез следене на шината за данни – процесорните звена комуникират чрез обща шина за данни. Процесорите наблюдават заявките на другите процесори. Общата шина е единна точка за синхронизация. Ако едно процесорно звено извика блок с права за запис, останалите процесорни звена виждат тази заявка и маркират тяхното копие на блока като невалидно. Предимството на този метод е ниската латентност при обновяването или

четенето от паметта, както и лесната изработка. Имаме само една транзакция директно към шината, без допълнителни околни пътища. Недостатъкът е общият ресурс, което прави трудно изпълнението при голям брой процесорни звена. За разлика от метода чрез указател, при следенето на шината не може да подаваме различни заявки едновременно.

Ако разгледаме базов протокол за обновяване на променливите с две състояния на блок от паметта – валиден и невалиден, като заявките за запис или четене върху блок се разпространяват по общата шина, тогава ще имаме четири възможности:



По общата шина разпространяваме:
PrRd – Processor Read – процесорът чете от локалната памет;
PrWr – Processor Write – процесорът записва в локалната памет;
BusRd – Bus Read – заявка за четене на локалната памет;
BusWr – Bus Write – заявка за запис на локалната памет.

Фиг. 2.3 Опростен протокол за обновяване на променливите с две състояния

(1) Процесор с валиден блок засича заявка от друг процесор за запис върху копие от този блок. Процесорът ще маркира неговия блок като невалиден.
(2) Процесор с невалиден блок изпраща заявка за четене от съседен кеш или оперативната памет за този блок. След прочитане маркира своя блок като валиден.
(3) Процесор с валиден блок изпраща заявка за четене или запис до останалите процесори. Заявката е информативна за останалите процесори и неговото копие остава валидно.
(4) Процесор с невалидно копие засича заявка за четене или запис от друг процесор. Тогава копието на процесора остава невалидно.

2.2.2.1 Апаратно съгласуване MSI

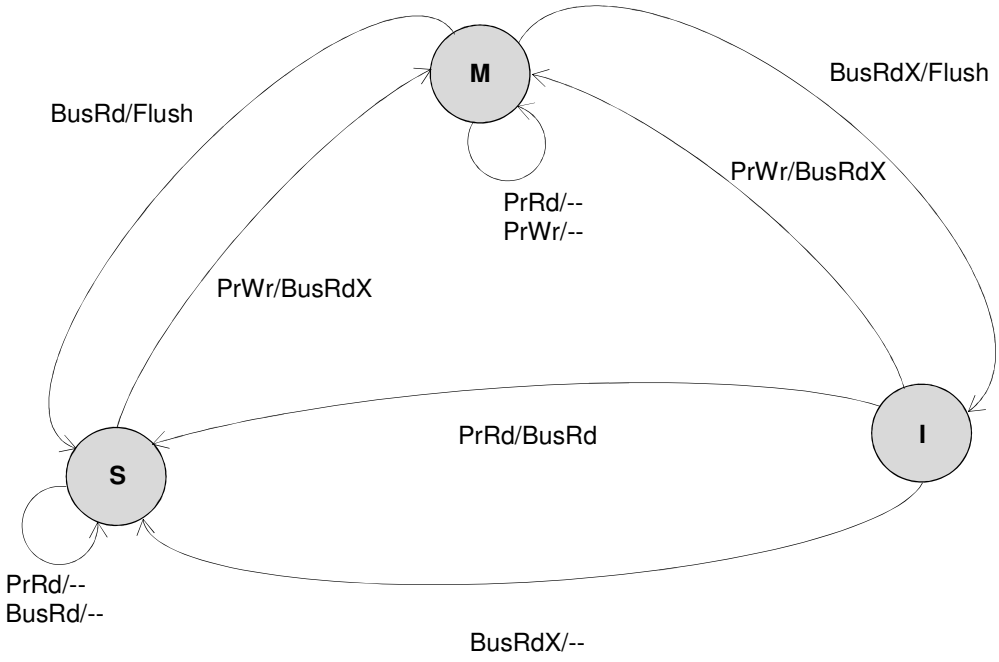
При MSI протокола, съкратено от Modified Shared Invalid, е добавено допълнително, трето състояние на блок от паметта.

M – модифициран – блокът от паметта е обновен в кеша и данните от блока са различни от останалите копия; контролерът на паметта има отговорността да обнови данните на копията на този блок.

S – споделен – блокът от паметта присъства в поне един кеш; контролерът може да изчисти блока от кеша, без да обновява оперативната памет.

I – невалиден – блокът не присъства в кеша и трябва да бъде извлечен от друго кеш звено или от оперативната памет.

Допълнителното състояние M – модифициран, позволява на процесора да обнови блок от паметта, без да разпространява информация по шината за неговото обновяване, когато блокът не е споделен.



Фиг. 2.4 Протокол за обновяване на променливите MSI

По общата шина разпространяваме:
Flush – информативен сигнал по шината за операцията;
PrRd – Processor Read – процесорът чете от локалната памет;
PrWr – Processor Write – процесорът записва в локалната памет;
BusRd – Bus Read – заявка за четене на локалната памет;
BusWr – Bus Write – заявка за запис на локалната памет.

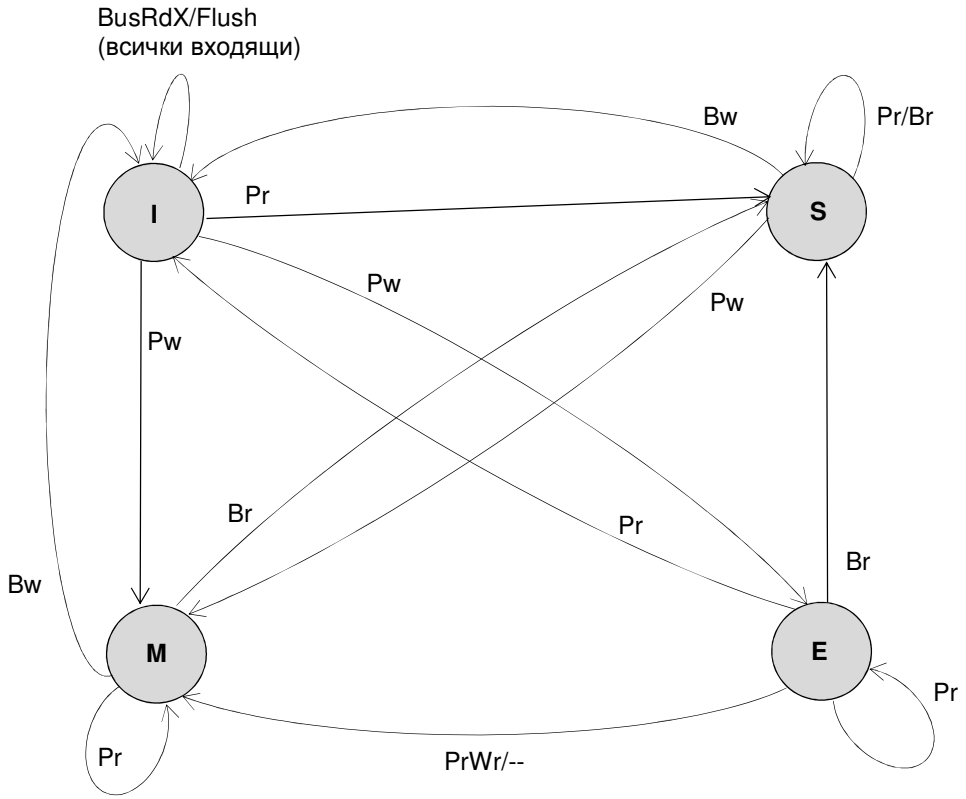
Когато процесорът даде заявка за блок от паметта, който липсва в кеша, контролерът извлича блока от оперативната памет и го маркира като споделен. Когато процесорът даде заявка за запис върху блок от паметта, който липсва в кеша или е споделен, маркираме блока като модифициран и разпространяваме сигнал, че блокът е

обновен. Чрез този сигнал маркираме всички останали копия като невалидни. Когато процесорът засече сигнал за запис върху блок от друг процесор, с присъстващо копие в неговия кеш, копието трябва да се маркира като невалидно. Предимството на MSI протокола е, че когато имаме блок със състояние M (модифициран), процесорът не трябва да разпространява сигнали по шината за четене или повторен запис.

Проблемът на този модел е, че когато процесор даде първоначалната заявка за блок, този блок се маркира като „споделен“. Ако същият процесор модифицира блока, то трябва да разпространи сигнал за анулиране на всички копия, дори този блок да се намира само в кеша на процесорното звено и да няма други копия. Това се дължи на факта, че процесорът няма информация дали други процесорни звена са извлекли този блок преди неговата промяна. Ако процесорното звено има информация, че притежава единственото копие на блока, би могло да направи модификация, без да разпространява сигнал за анулиране.

2.2.2.2 Апаратно съгласуване MESI и MOESI

Протоколът MESI, съкратено от Modified Exclusive Shared Invalid, е подобрение на протокола MSI. Повечето съвременни многопроцесорни системи имат версия на този протокол. При него добавяме още едно състояние, Exclusive – изключително, даващо права на процесора да обновява копието, без да разпространява информация за това. Поставяме блок от паметта в състояние E, когато при четене от оперативната памет блокът не присъства в друг кеш. Когато блок от паметта е маркиран в това състояние, процесорното звено, работещо с блока, притежава единственото копие и може да прави промени върху него, без да е необходимо да разпространява информация за това.



Фиг. 2.5 Протокол за обновяване на променливите MESI

Преход от състояние E или M към състояние S се нарича „понижение“, защото преходът отнема правата на процесора да модифицира блока без съобщение по шината. Съответно промяна от S (споделено състояние) към състояние E или M е „повишение“, защото дава права на процесорното звено за „безмълвна“ промяна на блока.

Когато понижаваме състоянието от E или M, възниква въпросът към кое състояние да го понижим. Може да маркираме блока като невалиден или като споделен. Ако решим да понижим блока към състояние S, това означава, че трябва да извлечем обновения блок. Така ще спазим условието за състояние S – блокът е споделен и притежава последната актуализирана стойност. Обновяването на всички копия в състояние S може да доведе до излишен трафик върху общата шина, при условие че обновените копия не се ползват.

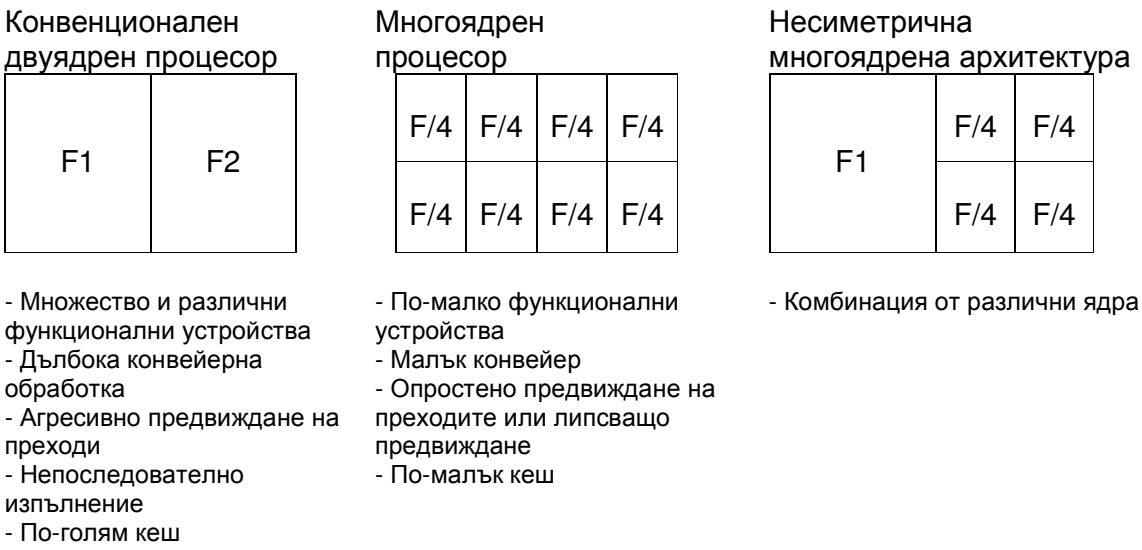
Едно от решенията е, когато понижаваме състоянието на блок от паметта, винаги да преминаваме към състояние I, т.е. маркираме останалите копия като невалидни. Така, ако някое от процесорните звена има нужда от обновения блок, само тогава ще даде заявка за обновеното копие. Това обаче ще доведе до излишни заявки за обновяване, които отново заемат от времето на общата шина. Друг недостатък е, че при честото използване на един блок от два процесора едновременно, обновените данни постоянно трябва да бъдат прехвърляни от един кеш към друг и обратно.

Друго решение е при промяна да обновим копията на блока към състояние S – споделено, но като посочим едно от звената като собственик (O – owner) на блока. Така разширяваме модела в протокол MOESI, съкратено от Modified Owner Exclusive Shared Invalid. Така блок в състояние S може да бъде с различни данни от последните обновени.

Протоколите могат да бъдат обновени с още състояния или чрез механизми за предвиждане, с които да намалим излишните заявки за инвалидиране на блок или трансферирането на обновен блок от паметта към неговите копия. Това допълнително ще усложни протоколите, защото ще преминаваме от едно състояние към друго не само при обновяване, а и при допълнителни условия, предвиждащи ползването на блока. Целта е да обновяваме блока само в памет, където този блок ще се ползва. Недостатъкът при допълнителните оптимизации е, че системата е по-трудна за проектиране. Появяват се грешни състояния на копията при състезания на сигналите. Добавянето на нови състояния или проверки не увеличава пропорционално производителността, а всяко ново състояние или проверка ще подобри малко предишния протокол, като в различни приложения може дори да предизвикат забавяне при изпълнението.

2.3 Несиметрични многоядрени архитектури

Съвременните процесори се делят предимно на едноядрени и многоядрени с еднотипни ядра [9]. Стандартно многоядрените системи са съставени от няколко или множество хетерогенни процесора с по-малка изчислителна мощност. Целта на многоядрените системи е да постигнем паралелизъм със степен броя на процесорните ядра, с което да подобрим цялостната производителност на системата. Както разгледахме по-горе, поради няколко причини не постигаме пропорционално увеличение на производителността с добавянето на нови процесорни звена. Причините са ползваните общи ресурси, възможностите за сегментирането на обработваното приложение, несиметричното натоварване на различните процесорни звена. При различни приложения сегментирането и обработването на отделните сегменти ще доведе до различни резултати, като понякога може дори да забави времето на изпълнение. Така последователна секция на програма с ниска степен на паралелизъм ще бъде изпълнена по-бързо върху едно ядро с по-добра функционалност. Паралелните сегменти на приложението могат да бъдат изпълнени на много и по-малки ядра, докато изчакват изпълнението на последователната секция. При конвенционален дизайн на ядро на процесор, в зависимост от функционалните възможности на ядрото, ще ползваме различна зона от силициевата подложка.



Фиг. 2.6 Несиметрични многоядрени архитектури

Ако изберем модел с един голям процесор, имаме висока производителност на приложенията, непозволяващи паралелизъм, но лошо изпълнение на няколко приложения паралелно или едно приложение, позволяващо паралелна обработка. Големите ядра са и енергийно неефективни. С учетворяването на зоната за изработка на едно ядро постигаме удвояване на производителността, но с това използваната енергия ще се увеличи четири пъти [11]. От друга гледна точка, многопроцесорна система с нискофункционални ядра забавя частта от приложението, което не може да бъде сегментирано, както и губи производителност при липса на предвиждане на преходите и слаба конвейерна обработка в тези ядра.

Тенденциите през последните години са свързани с разработка на процесори с основно ядро за изпълнение на частта от приложението, която трябва да бъде изпълнена последователно, и много малки процесорни звена за частите от приложението, които позволяват паралелна обработка или изпълнението на отделни приложения. Друг модел на несиметричен процесор е с ядра, позволяващи работа с различна тактова честота. Това ще ни позволи при необходимост да увеличим подаваната електроенергия и тактовата честота към конкретно ядро. Когато разглеждаме архитектури с множество процесорни елементи в паметта, ще имаме асиметрична архитектура. Съществуващите асиметрични микроархитектури и компилаторите за тях ще ни позволят внедряването на mPIM (множество от помощни процесорни елементи в паметта) да стане незабележимо за програмиста и операционната система. Чрез асиметричните архитектури може да подобрим серийното претоварване по следните начини:

- Приложения, непозволяващи паралелизъм, изпълняваме на голямото ядро;
- Сегментите на приложения, позволяващи паралелизъм, ще изпълним на няколко малки ядра;
- Основното ядро може да ползваме при дисбаланс на натоварването. Ако сегмент от приложение отнема твърде много време за обработка на едно от малките ядра, може динамично да прехвърлим неговата обработка върху голямото ядро;
- Основното ядро ползваме за изпълнение на предварително предвидените критични части от едно приложение. Критична част от приложение е част, изискваща ресурси или променливи, необходими на останалите сегменти за тяхното изпълнение.

За ефективността на системата е важно да откриваме местата, забавящи изпълнението на приложението. Това могат да бъдат различни сегменти от едно или много приложения, изпълнявани паралелно. Както споменахме, това са критичните сегменти на приложението, но също така могат да бъдат програмни бариери, забавяне в конвейерната обработка, комуникацията по шината към паметта, пропуски при четене от кеша и други. Целта е динамично да откриваме най-важните места, забавящи паралелната обработка, и да предаваме тяхното изпълнение към основното ядро. Софтуерно програмистът може да отбелязва бариери, като въвежда функция за извикване на изпълнение от основния процесор. При софтуерна бариера програмистът може да изпрати най-бавния сегмент до основния процесор и по този начин бързо изпълнените сегменти върху малките ядра няма да изчакват дълго неговото изпълнение. Компилаторът също може да открива и отбелязва критични секции, като въвежда функцията за извикване на обработка от основния процесор. Хардуерно динамично може да засичаме сегменти, реализиращи най-бавно изпълнение върху малките ядра. Тези сегменти автоматично изпращаме за изпълнение към основното ядро, без това да е видимо за програмиста или компилатора. Начинът, по който определяме тези сегменти, е чрез таблица за запис на времето, в което определено ядро е в процес на изчакване за определен сегмент.

Всеки път, когато процесорно ядро попадне в период на изчакване, записваме в таблицата изчаквания сегмент с неговия идентификационен номер, броя на малките ядра, изчакващи този сегмент, и броя на циклите, през които малкото ядро е в процес на изчакване. Ако друго ядро попадне в период на изчакване за изпълнение на същия сегмент, увеличаваме броя на чакащите ядра с едно и броят на циклите започва да се увеличава с броя на чакащите ядра за цикъл. Когато ядро приключи изчакването на заетия сегмент, изпраща сигнал до таблицата да прекрати увеличаването на броя цикли за чакане. След определен период по изпълнението на едно приложение ще запълним таблицата с различни сегменти и общия брой цикли, които те са изчаквали. Механизмът за ускоряване на процеса е на базата на чаканото време до момента. Когато едно малко ядро получи

сегмент за изпълнение, първо изпраща запитване до таблицата колко цикъла е изчакван сегментът до момента, и ако сегментът е с по-малък брой цикли на изчакване, се изпълнява от малкото ядро. На базата на предвиждане, ако до момента конкретният сегмент от приложението е чакан голям брой цикли, малкото ядро изпраща сегмента за изпълнение от основния процесор. Като подобрение, сегментите за изпълнение от основния процесор могат да бъдат приоритизирани също на базата на броя чакани цикли. Хардуерно цената за изработка не е много висока. За този тип допълнение към процесорната архитектура ще ни е необходима памет за таблицата на сегментите и таблица за приоритизиране на изпълнението в голямото ядро. Недостатъкът при този тип обработка от несиметрична многоядрена архитектура се получава при грешно предвиждане на критична част от приложението, както и от липсата на повече от едно голямо ядро. Внедряването на изчислителни ресурси в паметта ще ни позволи създаването на архитектура от малки хомогенни процесори директно в паметта и многоядрен основен процесор. Няколко сегмента от критичните части на приложението ще могат да се обработват едновременно от ядрата на основния процесор, докато останалите се обработват направо в паметта. Това ще ни предостави широк обсег за ускорение на изпълнението на частите от приложение, които не могат да бъдат сегментирани, като изпълнението е изцяло от хардуера, без усилия от програмиста или компилатора.

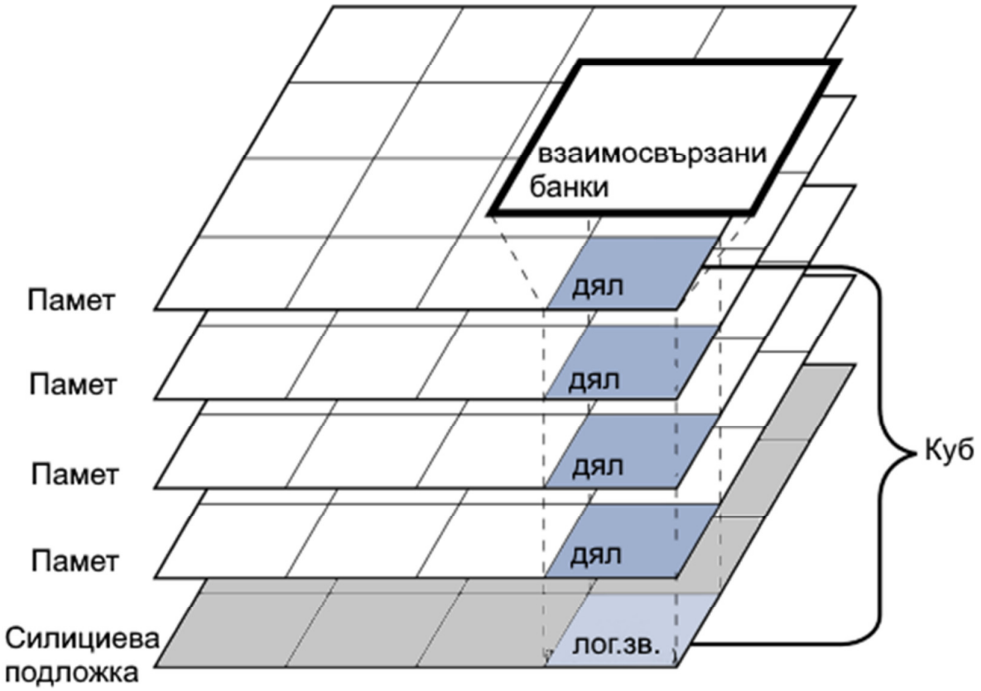
2.4 Изследване и предложения за използване на mPIM

2.4.1 Недостатъци при PIM и mPIM

Основният недостатък за развитието на процесорните звена, близо до паметта, е разликата в използваната технология. Производството на кондензаторните елементи на паметта се различава от производството на силициевата подложка за процесорните елементи. Ако искаме да използваме един тип технология, основната памет трябва да бъде изградена като SRAM. По този начин ще може да произведем изчислителните елементи и паметта от една силициева подложка. Този тип производство е възможен, но ще бъде изключително скъп поради производствената цена на SRAM паметта. Ако използваме стандартната DRAM памет, ще имаме комбинация от кондензатори и логични елементи. Това няма да ни позволи да използваме един елемент за процесорните елементи и паметта едновременно.

През 2011 Samsung и Micron (HMC Tech – Hybrid Memory Cube Tech) започват обща работа върху разработката на нов тип хибридна 3D памет [8]. Към консорциума по-късно се включват Microsoft, ARM, HP, SK Hynix и Fujitsu, като той се преименува на

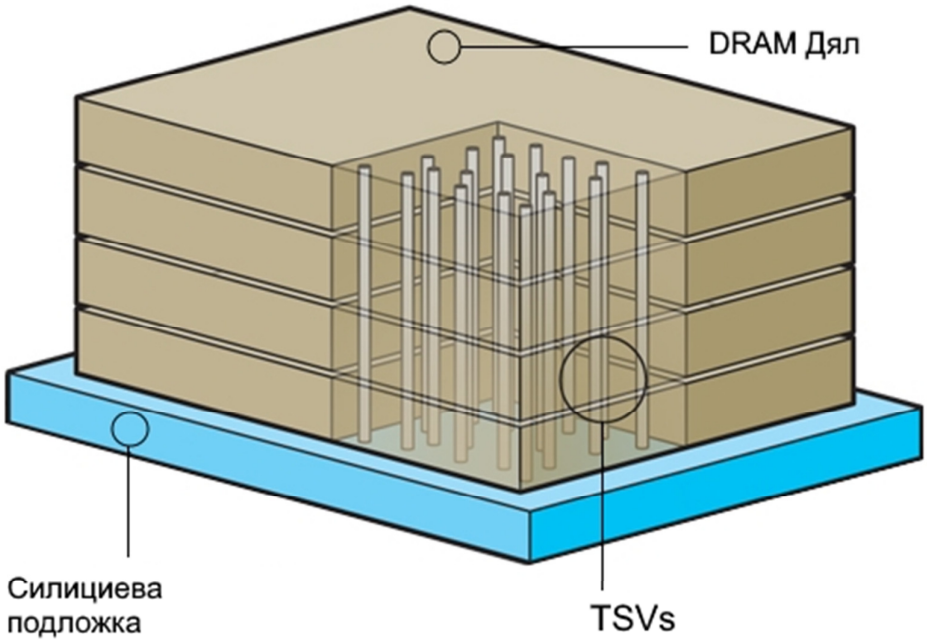
HMC Consortium. Триизмерната хибридна технология „Hybrid Memory Cube“ е съставена в единична опаковка, съдържаща няколко слоя DRAM памет, изградени директно върху силициева подложка с логични елементи. Различните слоеве са свързани чрез TSV (through-silicon via) технология. TSV технологията позволява вертикална взаимовръзка, преминаваща и свързваща силициевите подложки или интегрални схеми. Вертикалната връзка позволява подложките да са една над друга, което ни дава повече памет на по-малко пространство, както и по-добра свързаност между слоевете. При триизмерните пакети, свързващи интегрални схеми, в повечето случаи взаимовръзките са изградени през техните ръбове. При хибридния куб вертикалните връзки са изградени вътре в пакета, което драстично съкращава електрическите пътеки. Примерна архитектура на организацията на хибриден куб може да бъде:



Фиг. 2.7 Примерна HMC архитектура

Примерният модел, предоставен от консорциума, разглежда разделението на подложките на симетрични дялове. Дяловете, които са един над друг, се разпределят на общ куб, управляван от дела на логичната подложка под тях. При този модел всеки куб има собствен контролер в логичната подложка, който управлява всички заявки до паметта в рамките на куба. Всеки контролер на куб определя операциите към куба независимо от останалите. Контролерите могат да имат собствени буфери, като различни модели позволяват заявките към куба да се определят от контролера, а не по реда на подаване.

Хибридната технология позволява разработването на изчислителни ядра, изградени върху първата силициева подложка.



Фиг. 2.8 Свързване чрез TSV (through-silicon via) технология

Примерните и тествани изпълнения, предложени от консорциума, включват:

- Всички входно-изходни операции да бъдат изпълнени като сериализирани пълен дуплекс връзки за отделните кубове;
- Индивидуални контролери за всеки куб с индивидуално управление на маршрутизацията на данните и буфериране на входно-изходните връзки;
- Конфигурируеми регистри и функции за управление на паметта.

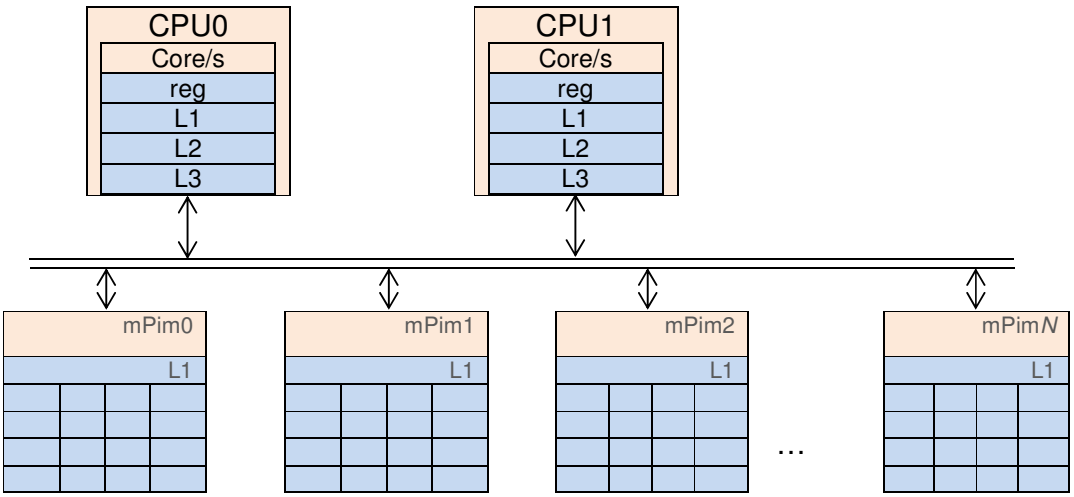
Пакетът включва общата вътрешна шина за комуникация между кубовете, която е свързана към входно-изходните връзки. Цялата комуникация е пакетизирана. Всеки пакет определя една пълна инструкция. Например заявка за четене, изпратена към хибридния куб, ще се състои от един пакет и отговорът на заявката може да бъде върнат в различен ред от другите подадени заявки. Това се дължи на факта, че всяка заявка може да бъде обработена от различен контролер на куб. Кубовете реорганизират техните пакети с операции за оптимизиране на наличните ресурси и за намаляване на закъсненията. Всички тези предложения понастоящем са внедрени и тествани, но не са ограничени като добавяне на допълнителни ресурси, и позволяват различните елементи да бъдат допълнително преконфигурирани.

Страницирането и обновяването на различните страници от различните процесори е друг проблем, с който се сблъскваме, когато имаме изчислителни ресурси в паметта. Изместването на контролера на паметта към самата памет при HMC технологията

едновременно с това ще разреши и този проблем. За момента, след промяна в местоположението на дадена страница централният процесор прави заявка към контролера на паметта за опресняване на страницата. Релокализацията на страницата се извършва на ниво памет. Всички заявки за обновяване, изпратени от изчислителните ресурси в паметта, могат да бъдат изпратени към слоя с контролера на паметта.

2.4.2 Предложение за архитектура с използване на mPIM

С времето става все по-трудно да се изработва едно голямо монолитно процесорно ядро поради сложността на изпълнението и консумираната електроенергия. Фокусът на индустрията се измества от едно монолитно ядро върху чип към многоядрени процесори, съставени от няколко хомогенни ядра върху една силициева подложка и многопроцесорни архитектури. Самите системи от своя страна могат да бъдат силно свързани или слабо свързани. Силно свързаните системи ползват обща шина за данни, по която комуникират при работата върху общ проблем или споделена памет, в която записват инструкции и данни по общ проблем на работещите процесорни ядра. Слабо свързаните системи комуникират през висикоскоростни мрежи за комуникация. Методите за заемане на ядрата варират от заемането от всяко ядро на различни задачи до разделянето на една задача на различни сегменти, които да бъдат изпълнени едновременно. За подобряване изпълнението на една програма върху многопроцесорен компютър е необходимо приложението да бъде разделено на различни сегменти, които да бъдат изпълнявани паралелно. При изследването на методите за ползване на mPIM ще се съсредоточим върху архитектура, съставена от основен конвенционален процесор или няколко процесора и много хомогенни малки процесорни ядра, разположени в паметта. Това ще е силносвързана архитектура, ползваща обща шина и памет за комуникация. Наличието на основен конвенционален процесор ще ни позволи да ползваме всички текущи разработки върху паралелизиране на програмите. Фокусът на предложените подобрения е да постигнем подобрения, без да се налагат допълнителни промени на програмите или тяхното прекомпилиране, като това остане скрито за програмиста. Основният процесор ще ни позволи също така да възлагаме различните сегменти върху ядрото, което ще ги изпълни най-ефективно. Интегрирането на процесорни ядра в един чип с паметта позволява ниска латентност и бърза комуникация между процесорите и паметта. Изработването на малки хомогенни процесорни елементи е рентабилно, като също така ще намали консумираната електроенергия при работа на процесорите.



Фиг. 2.9 Многопроцесорна архитектура с изчислителни ресурси в паметта

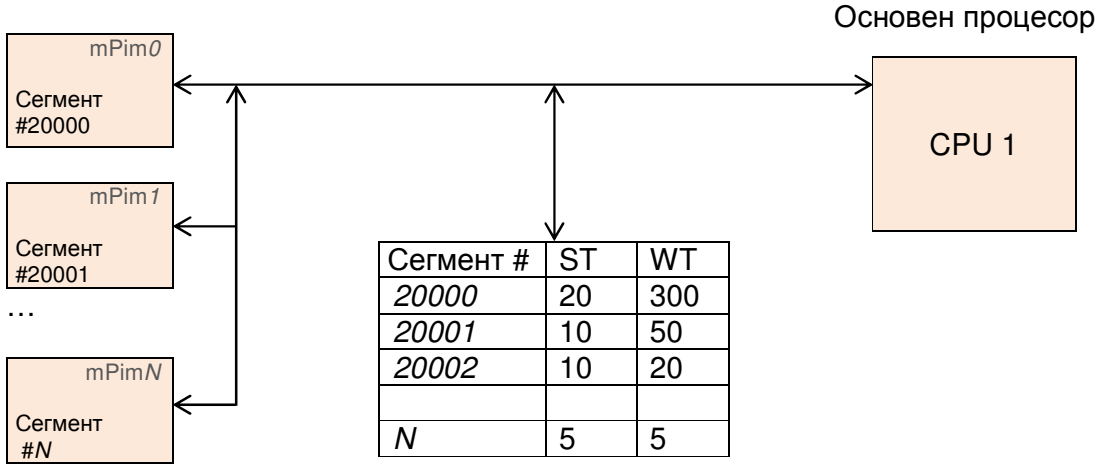
Този клас хетерогенна архитектура е съчетание от един или няколко основни процесора с голям брой инструкции и кеш нива с висока латентност при заявки до паметта, и набор от процесорни елементи в паметта с малък комплект инструкции, но по-ниска латентност до основната памет.

- Паралелна обработка на сегментите от mPIM и изпълнение на последователната порция код от основния процесор

Различните приложения позволяват различни нива на паралелизиране. Както разгледахме по-горе, програмите, позволяващи паралелна обработка, са най-ефективни при многопроцесорни архитектури, но въпреки това и те имат своите ограничения. Общото време за изпълнение на едно паралелизирано приложение ще зависи както от броя изчислителни ядра и броя сегменти, така и от времето за изпълнение на най-бавния сегмент.

Динамичното идентифициране на непаралелизираната порция код и изпращането на тази порция към основния процесор ще бъде критично за цялостното време на изпълнение на приложението. Динамичното идентифициране на непаралелизираните части от кода може да се извършва чрез хардуерно-софтуерно решение на ниво системни инструкции и микроархитектура, скрито от програмиста. Това ще се извършва, като при изпълнение на паралелизирано приложение измерваме времето за изпълнение на всеки сегмент. Всеки сегмент се идентифицира с уникален номер. Заделяме таблица в паметта, в която да записваме броя цикли, за който е изпълнен, както и евентуални зависими сегменти, изчакващи неговото изпълнение. При началното изпълнение на една програма всички сегменти се изпълняват от изчислителните ресурси в паметта. За всеки цикъл

процесорният елемент в паметта увеличава ST клетката в таблицата с номера на сегмента, който изпълнява. Ако има процесорно ядро, изчакващо този сегмент поради зает общ ресурс, стойността в клетката WT, асоциирана със сегмента, се увеличава също. Когато имаме повече от едно процесорно ядро, изчакващо определен сегмент, тогава за всеки цикъл ще увеличаваме WT с броя на изчакващите ядра (Фиг. 2.10). Този метод ни позволява не само да следим най-бавния сегмент от една програма, но и сегменти, които заемат общи ресурси и не позволяват други сегменти да бъдат изпълнени.



ST – Segment Time – Време за изпълнение на сегмента
WT – Wait Time – Комбинирано време на изчакващи сегменти

Фиг. 2.10 Многопроцесорна архитектура с изчислителни ресурси в паметта

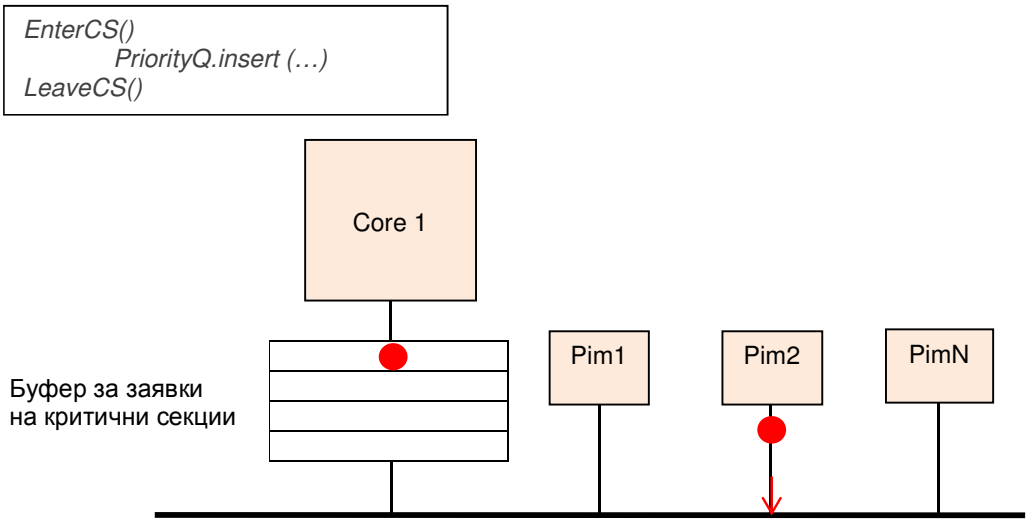
Когато имаме достатъчно информация за времето и ползваните ресурси на едно приложение, ние можем да взимаме решения кои сегменти да бъдат изпратени към основния процесор. Таблицата е динамично обновявана, като решението за сегмента, изпращан към основния процесор, ще се променя с времето. Моделът е независим от броя ядра на основния процесор. Това прави системата удобна за разширение. Операционната система може да вземе решение да задели едно или няколко от ядрата на основния процесор за изпълнението на избраните сегменти от програмите.

- Паралелна обработка на сегментите от mPIM и изпълнение на критичните секции от основния процесор

Програмите, които позволяват сегментация, се изпълняват на различни ядра, работейки върху общ проблем, като комуникират чрез споделена памет. За да се гарантира коректността на изпълнението на приложението, множеството сегменти не могат да актуализират споделените данни едновременно. Този метод за изпълнение е известен като принцип на взаимното изключване. Прилага се, като достъпът до споделените данни е заключван в региони от код, синхронизиран чрез примитиви за

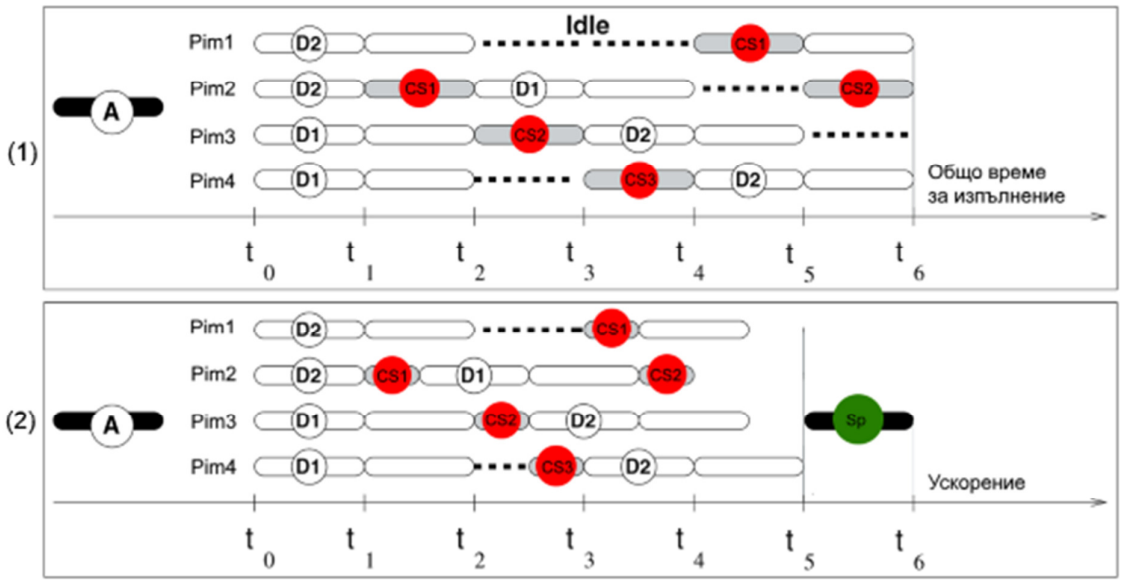
синхронизация на данните. Тези заключени региони от код се наричат критични секции на паралелизираното приложение. Само един от сегментите на приложението може да изпълнява критична секция в определен момент. Всеки друг сегмент, изискващ достъп до споделените данни на критичната секция, трябва да изчака завършването на текущия сегмент. По този начин, когато имаме няколко сегмента, изчакващи критична секция, заета от друг сегмент, ще получим цялостно забавяне в изпълнението на приложението. С увеличението на сегментирането на едно приложение се увеличават конфликтите за достъп до критичните секции.

При традиционните многоядрени системи, когато едно ядро достигне до критична секция, то дава заявка за заключване на секцията. След изпълнение на критичната секция ядрото я освобождава. При модела за изпълнение от ядрата в паметта, след като едно ядро попадне на критична секция, ядрото я заключва и праща заявка към основния процесор за изпълнение. Основният процесор усвоява критичната секция и след нейното изпълнение уведомява ядрото в паметта. Това ще ни освободи при програмирането на паралелизирани приложения да се разработва код с по-големи критични секции за избягване на грешки, без да се грижим за времето за тяхното изпълнение. За изпълнението основният процесор може да има нужда от определени данни от ядрото в паметта. Тези данни се прехвърлят към основния процесор от кеша на ядрото в паметта със стандартните налични механизми за съгласуване на кеша, разгледани по-горе. Трансферът на данни към основния процесор би могъл да забави изпълнението на приложението. В случаите, когато сме заделили основния процесор за изпълнение само на критичните секции, трансферът на данни по общата шина ще е нисък и прехвърлянето на критичните секции ще отнеме стандартното време за прехвърляне на данни от оперативната памет, без допълнително време на изчакване.



Фиг. 2.11 Механизъм за ускорение на критичните секции

При достигане изпълнението на критичната секция, с други думи, когато процесорът достигне до изпълнение на примитив *LockX,* на ниво системни инструкции отбелязваме навлизането в критична секция *EnterCS()*. Кодът на критичната секция се изпраща към буфера на основния процесор и ядрото в паметта изчаква неговото изпълнение. След изпълнението на кода от основния процесор критичната секция се изключва от основния процесор и резултатът се връща към ядрото в паметта, което продължава изпълнението на програмата.



Фиг. 2.12 Разпределение на критичните секции към основния процесор

(1) Стандартно изпълнение от многопроцесорна архитектура:

Core N

```
A = compute ()
LockN
        Execute_A
        …
        …
        …
        …time…
        …for…
        …small…
        …core…
        …
        …
        …
        result=CS(A)
UnlockN

print result
```

(2) Изпълнение чрез mPim и критичната част на същия код от основния процесор:

mPim N

```
A = compute (),
LockN
  PushA
  CallCS(N),PrCounter++
  …
  …
  …
  …
print result

Ускорение
```

Main CPU

```
...
...
Aquire_A
Execute_A,
...time…
RetCS(N), UnlockN
...
```

Фиг. 2.13 Хардуерно изпълнение на разпределението към основния процесор

От друга гледна точка, изпълнението на критичните секции изключително и само от основния процесор може да доведе до проблем при висока степен на сегментация на приложението. Критични секции, заключени чрез примитиви с независими ключове, реално биха могли да бъдат изпълнени паралелно. Изпълнението на всички критични секции само от един процесор налага последователно изпълнение и не позволява паралелизъм. Отделни механизми могат да взимат решения за селективното изпълнението на критичната секция от ядрото в паметта. Динамичното решение за изпращане на критичните секции към основния процесор се изпълнява на ниво системни инструкции, като и това остава скрито за програмиста. Метод за избирателност е осъществен при избора за изпълнение на две независими критични секции. Ако в целия буфер на основния процесор имаме заредени няколко зависими критични секции, а едно от ядрата в паметта попадне на критична секция, независима от заредените, то ние можем да започнем нейното изпълнение паралелно в mPIM. Така ще постигнем паралелизация при изпълнението на независимите критични секции по същия начин на изпълнение, както при конвенционалните многоядрени архитектури. Необходимите промени ще са изключително малко и на ниво системни инструкции. Ще е нужно да добавим две нови инструкции – *CallSC* и *RetCS*. Инструкцията *CallSC* е подобна на традиционната инструкция *Call*, но изпълнява кода дистанционно. Ядрото в паметта изпраща инструкцията към основния процесор и изчаква резултата от нейното изпълнение. По същия начин *RetCS* е подобна на традиционната *Ret* инструкция, с изключение на това, че резултатът се връща след дистанционно изпълнение. Изпращането на данните ще изпълним със съществуващите методи за съгласуване на паметта. Промените на операционната система са свързани с разпределението на ядрата. Когато основният процесор е едноядрен, тогава ядрото се заема да изпълнява последователно буферираните критични секции и не се заема със заявки за изпълнение на други сегменти. Когато имаме няколко работещи паралелни програми, операционната система заделя основното ядро през равни интервали от време за конкурентно изпълняваните програми. При архитектури с многоядрен основен процесор, операционната система може да задели само едно или няколко от ядрата на основния процесор, като използва останалите конвенционално. В ситуациите, когато имаме няколко работещи паралелни програми, операционната система може да заделя основното ядро през равни интервали от време за конкурентно изпълняваните програми. Като резултат кодът за изпълнение и съдържанието на една програма остават непроменени. Не са необходими специални модификации, тъй като съвременните операционни системи поддържат многоядрени архитектури и специализирано разпределение на различните ядра.

Предимството на тази архитектура е бързото изпълнение на критичните секции. Зависимите примитиви за заключване са централизирани и се изпълняват последователно от основния процесор, което отхвърля възможността за срив при грешно паралелизиране на различните критични секции. Недостатък е забавянето от времето, необходимо за прехвърляне на данните през шината на паметта. Основният процесор ще бъде зает изключително и само за изпълнение на критичните секции от програмите, работещи върху ядрата в паметта, което ще забави изпълнението на последователните части от различните приложения. Този проблем е лесно разрешим, ако имаме многоядрен основен процесор. Тогава едно или няколко от ядрата на основния процесор ще бъдат заделени за обслужване на критичните секции. Останалите ядра на основния процесор ще са заети с паралелните части от приложенията или с изцяло различни приложения. Важно е да отбележим как системата ще приема системните прекъсвания IRQ (interrupt request). Прекъсванията към основния процесор се обработват по конвенционалния начин. При подаване на системно прекъсване основният процесор преустановява изпълнението на критичната секция. Заявките за системните прекъсвания към изчислителните елементи в паметта се обработват приоритетно. Трябва да обърнем внимание при системно прекъсване, когато процесорният елемент е в процес на изчакване на резултат от основния процесор. Функцията от основния процесор за връщане на резултат *RetCS* се обработва като приоритетно системно прекъсване от процесорния елемент. Всички останали системни прекъсвания в моменти, различни от изчакване на резултат от основния процесор, се третират чрез конвенционалните механизми.

- Използване на PIM за предварително зареждане на кеша и намаляване на студени пропуски

Презареждането на данни в кеша на основния процесор, изпълняващ програмата, се базира изцяло на предварителна обработка на същата програма. Методите на този модел могат да бъдат два:

- Изпълняване на парче от основната програма с цел презареждане на извикваните променливи;

- Изпълняване на съкратена програма със същата цел: съкратената версия на програмата може да бъде генерирана от компилатора независимо от програмиста, както и от самия програмист.

При тези методи ние трябва да изпълним отделни сегменти или съкратена версия на програмата, които водят до пропуски при заявки към кеша. Тази част от кода изпълняваме върху PIM ядро и ще я разглеждаме като отделна спекулативна нишка от програмата. През времето, когато основното ядро изпълнява началната фаза на една

програма и продължава през пълното ѝ изпълнение, PIM ядрото ще работи върху спекулативната нишка. По този начин PIM ядрото, изпълняващо спекулативната порция от приложението, ще зареди необходимите данни за основното ядро. Данните ще са заредени за оригиналната програма, преди тя да е достигнала до тяхното изпълнение. Така, когато оригиналната програма стигне до заявка на данни, те вече ще са заредени в кеша и няма да има нужда основното ядро да изчаква тяхното зареждане. Основният проблем при тези методи е как да конструираме спекулативната нишка.

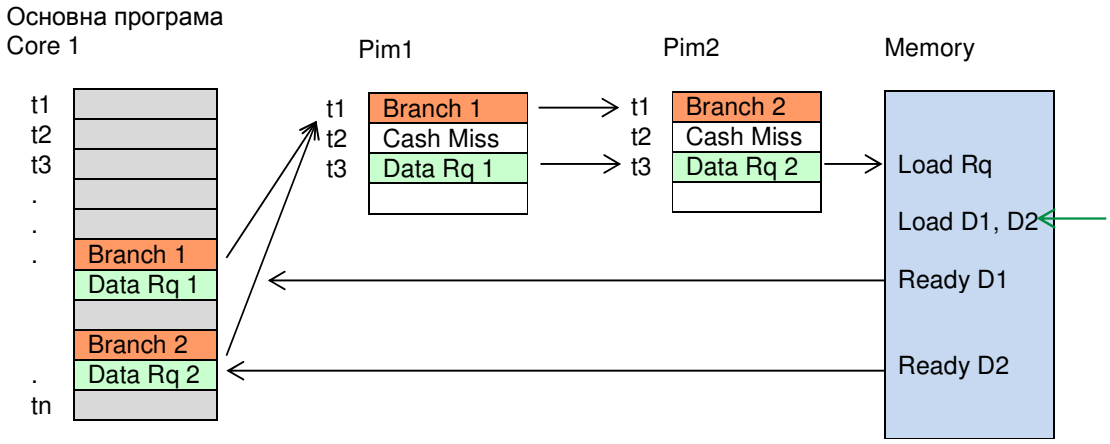
Един от начините е да постигнем това чрез софтуер. Самият компилатор може да генерира спекулативната нишка и когато основната програма достигне определен етап на изпълнение, спекулативната порция код се зарежда в свободно ядро от паметта. По същия начин спекулативната порция код може да се извика и от програмиста след достигане на този етап от оригиналната програма, но това ще изисква промяна в кода на приложенията, приготвени за конвенционални системи. Друг метод е чрез хардуер, като описаните по-горе функции могат да се изпълнят единствено и само от него. Това е постижимо, но ще е трудно изпълнимо. Хардуерно трябва да конструираме самостоятелно спекулативната нишка за изпълнение. Трети метод е като използваме оригиналната програма, която да изпълняваме по-бързо. Тук веднага възниква въпросът как по-малко и по-бавно ядро от паметта може да изпълни основната програма преди основния процесор. Начинът е като изпълняваме програмата без забавяне поради незаредени данни, без проверки за грешки и без да отдаваме значение на крайния резултат от това изпълнение.

От изброените методи виждаме, че всички имат определени предимства и недостатъци. Основният недостатък е използването на допълнителни ресурси за изпълнението на спекулативна част от кода, която не е нужна за крайния резултат. Допълнителен недостатък е, че в зависимост от избрания метод презаредените данни може да се окажат грешни. При зареждане на грешни данни ние няма да нарушим изпълнението на основната програма, но времето за обработка ще бъде като при конвенционална система. Рискуваме да отхвърлим полезни и необходими данни от паметта за сметка на грешно заредените. Основното предимство е бързото изпълнение на основната програма. Този тип предвиждания може да се използва от система, при която ползваните ресурси не са приоритет, но ни е необходимо бързо и коректно изпълнение на основната програма.

- Използване на PIM за предвиждане на преходи

При използването на PIM за предвиждане на преходи прилагаме метод, много близък до метода за предварително зареждане на данни в кеша. За предвиждане на преходите изпълняваме предварително части от програмата върху ядрата в паметта като

отделни помощни нишки. Всеки път, когато имаме разклонения, ние взимаме част от кода преди самите разклонения и го изпълняваме върху PIM ядрата. Отсетите сегменти се изпращат към помощните ресурси в паметта, като разчитаме, че след изпълнение на прехода ние ще заредим правилните данни за прехода в оперативната памет. По същия начин данните ще са заредени в оперативната памет за оригиналната програма, преди тя да е достигнала до тяхното изпълнение [2]. Тук няма да имаме нужда от конструиране на спекулативните нишки, защото може хардуерно автоматично да се избира фиксиран брой стъпки преди разклонението за обработка. Изпълнението ще се извършва изцяло без намеса от програмиста или промяна в компилатора. Основният проблем ще бъде времевата синхронизация, или по-точно – кога да изпратим спекулативните нишки за изпълнение. Ако изпратим спекулативните части за изпълнение твърде рано, може да заредим грешни условни променливи и да стигнем до грешно заключение при изпълнението на разклонението. Ако изпратим спекулативната нишка твърде късно, конвенционалният процесор може да стигне до разклонението, преди да сме го изпълнили в помощното ядро. Разрешение на този проблем може да се намери при симулация и статистическо изследване на времето, необходимо за изпълнение при спекулативно зареждане в PIM.



Фиг. 2.14 Ускорение чрез предвиждане на преходи

Голямото преимущество на спекулативното изпълнение на самите преходи в паметта е, че може да се извършва изцяло хардуерно. Няма да е необходима никаква промяна върху операционната система, компилатора или кода на програмата. Това е особен плюс, ако искаме да въведем PIM в съществуващи конвенционални компютри, без да правим каквито и да било промени по текущите разработки или работещия софтуер. Недостатъкът е използването на ресурси за изпълнение на сегменти, които после ще се

преизчисляват от основния процесор. Това няма да е проблем при енергонезависими системи, целящи скоростно изпълнение на задачите.

- Допълнителни и независими начини за използване на PIM

Както разгледахме по-горе, основният проблем при забавянето на съвременните процесори идва от претоварването на шината за пренос на данни. Проведени тестове показват, че на всеки четири такта изчислителното устройство губи три такта в изчакване на данни от паметта. Споделената шина до паметта от инструкциите на една програма с операндите на същата програма води до конфликт за достъп до ресурса, или до така наречения проблем „Von Neumann bottleneck“. Многопроцесорните и многоядрените системи задълбочават този проблем, защото отделните ядра се обръщат към споделения ресурс. Наличието на изчислителни ресурси в самата памет ни позволява както изпълнението на приложенията в самата памет, така и допълнителни възможности за намаляване на описания проблем.

Използването на PIM ядро за компресиране на данните, изпращани по шината на паметта, ще намали времето за достъп до шината, необходимо при заявка за изпращане и получаване на данни, което ще я освободи за другите ядра. Необходимостта от компресии на цифрови сигнали, с цел намаляване на използваните ресурси в областта на радиото и телевизията, е довела до разработването на множество алгоритми. Като цяло повечето алгоритми са ориентирани за компресиране на цели файлове, но също така има и алгоритми, разработени за еднопосочно компресиране на данни в момента на предаване. Като пример за такъв алгоритъм е компресирането с MPEG-2, широко използван за предаване на цифрови данни. При този алгоритъм предаваните данни не трябва да са цялостни, за да бъдат изпратени. Данните се разбиват на отделни блокове и всеки блок се компресира самостоятелно. Размерът на блоковете ще бъде от изключителна важност, като това е компромис между процента на компресия и размера. По-големите блокове ще имат по-добра компресия, но това означава и по-дълго време за изчакване от основния процесор. Основният процесор ще трябва да изчака целия блок да бъде предаден, преди да го разкомпресира и обработи данните. Компресирането на данните, освен всичко друго, ще ни позволи и да извършваме по-добър контрол над грешките. Самите пакети могат да съдържат CRC – контролна сума за проверка на цикличния остатък. Също така при откриване на грешка в определен пакет няма да е необходимо да изпращаме всички данни отново, а само пакета с грешки. В другия край на паметта може да имаме на борда декодер или да ползваме конвенционалния процесор за разкомпресиране. Разбира се, може да видим, че основният недостатък на този подход е допълнителното забавяне при компресирането и декомпресирането на пакетите. Методът няма да бъде подходящ при

изпращането на малки данни за обработка, защото това би довело само до тяхното забавяне към процесора. Отново хардуерно може да взимаме решения кога да компресираме данни и инструкции спрямо техния размер. Когато данните надвишат предварително фиксиран размер, те ще бъдат изпращани за компресия.

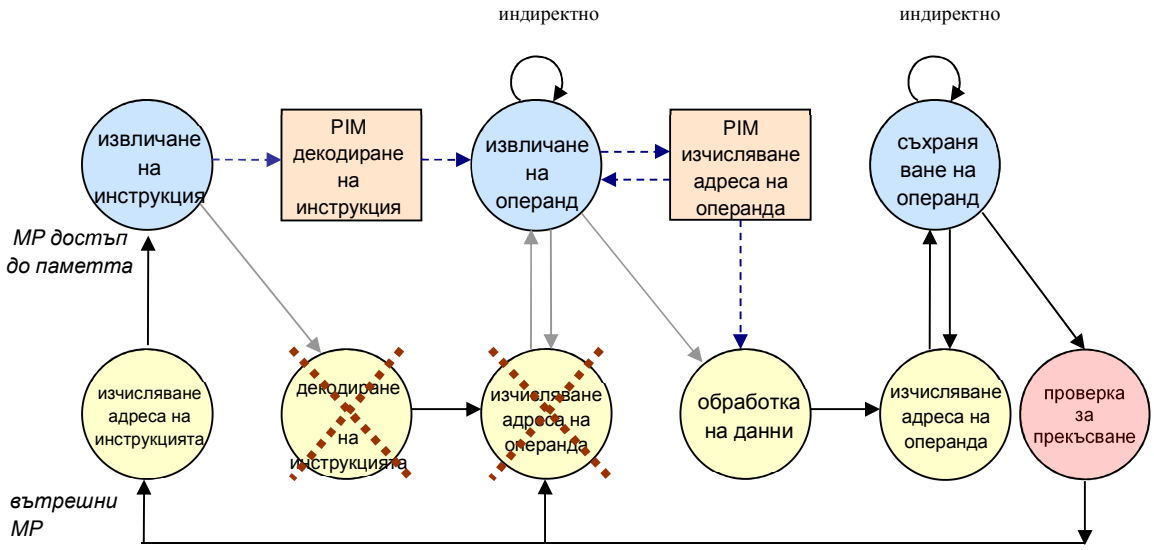
Освен компресирането на данните, ние може да ги изпращаме като отделни пакети, в смисъл на основно звено за комуникация през общата комуникационна шина. Както разгледахме по-горе, разделянето на пакети има преимуществото при компресия и проверка на данните, но също така пакетирането може да се използва за ускоряване на самата комуникация. Един пакет, освен суровите данни за пренос, може да носи в себе си допълнителна полезна информация. В началната част на пакета може да сложим адрес на получателя. По този начин, ако имаме комутационна връзка от паметта до процесорните ядра, ние ще може да активираме комутаторите без допълнителни усложнения и сигнали. Така типът връзка между процесорните ядра и паметта от тип шина може да бъде конвертирана като комутаторна мрежа. Промяната от тип обща шина към комутаторна мрежа ще ни позволи няколко процесорни ядра да комуникират с различни елементи памет по едно и също време.

Предимството на тези методи е отново липсата на необходимост от промяна на софтуера и компилатора. Всичко се извършва на ниво хардуер и е невидимо за операционната система или компилатора.

2.5 Аналитичен модел на използването на изчислителни ресурси в паметта

На Фиг. 2.15 по-долу сме изобразили диаграмата на инструкционния цикъл в PIM архитектурата. Основната разлика от инструкционния цикъл при класическата Von Neumann архитектура е, че тук вмъкваме 2 нови блока, в които се извършва предварителното декодиране на инструкцията и се изчислява адресът на операнда. И двете функции се изпълняват от PIM модула, като по този начин се изключва обръщението от основния процесор към оперативната памет и съответно се намалява потокът информация помежду им. От диаграмата се вижда, че при използването на PIM модул се редуцира неколкократно заявяването на шината на паметта (изобразено със сиви стрелки). Това намалява времето за комуникация между основния процесор и паметта, което увеличава цялостната производителност на системата. Забавянето в класическите компютърни архитектури, дължащо се основно на техническите характеристики на оперативната памет, а именно латентността, както и непрекъснатото нарастване на производителността на процесорите (виж Фиг. B.1), създава ограничаващ фактор, който предизвиква централния процесор да чака до извличане на инструкцията и операнда.

С използване на PIM за поемане на част от функциите на процесора това забавяне се редуцира, като PIM работи изцяло на ниво памет, показано в диаграмата на инструкционния цикъл. Чрез разделянето на инструкционния цикъл на две нива ясно може да се определи разпределянето на функциите между основния процесор и PIM.



Фиг. 2.15 Диаграма на инструкционния цикъл на PIM архитектура

Последователността от стъпки на ниво памет се поема изцяло от PIM възела, който извършва следното:

- предекодиране на суперскаларни инструкции;
- администриране и извличане на операндите;
- управление на политиката за запис (write-back и write-through);
- реализация на алгоритмите за заменяне (LRU, LFU и FIFO);
- решаване на зависимостите при запис и четене на данни.

От гледна точка на ниво основен процесор имаме изпълнение на следните функции:

- изчисляване на адреса на инструкцията;
- обработка на данните;
- решаване на зависимостите (четене след запис, запис след четене);
- преименуване на регистър;
- предсказване на преходите;
- преподреждане на записите.

Разбира се, при проектиране на нови видове компютърни архитектури е възможно част от тук изброените функции на PIM възела да бъдат прехвърлени за изпълнение от

основния процесор и обратно, което обуславя и архитектурната особеност на моделите, подробно разгледани в Глава 2.4.2.

За предварителна оценка на архитектура с множество PIM възли с общ брой от „N“ елементи е предложен аналитичен модел, описващ комбинацията на параметрите, представени подробно по-горе. Този аналитичен модел ще послужи по-нататък при проектиране на модела за предварителното определяне на основните параметри. В предложения модел се предполага, че за приложения с интензивен поток от данни, в които нямаме или рядко имаме повторно използване на определен сегмент от вече преминалия поток данни и малко налично количество кеш памет, PIM архитектурата може да бъде от голяма полза. Резултати от предложения модел се получават чрез симулация и са анализирани в Глава 4 за проверка на хипотезата, направена от аналитичното описание на предложената архитектура.

$$T = 1 - \%W_{PIM} \times \left\{ 1 - \frac{1}{N} \times \left[\frac{\tau_{pim} + LS \times \left(T_{Mpim} - \tau_{pim} \right)}{1 + LS \times \left(T_{Cmp} - 1 + P \times T_{Mmp} \right)} \right] \right\} \tag{2.5}$$

ако приемем, че $M \equiv \left[\frac{\tau_{pim} + LS \times \left(T_{Mpim} - \tau_{pim} \right)}{1 + LS \times \left(T_{Cmp} - 1 + P_{MP} \times T_{Mmp} \right)} \right]$ (2.6)

то за T получаваме $T = 1 - \%W_{PIM} \times \left\{ 1 - \frac{M}{N} \right\}$ (2.7)

където:

T = времето за пълен цикъл на PIM операция

$\%W_{PIM}$= процентно извършена работа от PIM

τ_{PIM} = такт на PIM

T_{Mpim} = PIM достъп до паметта

LS = средно време за четене и запис

T_{Cmp} = време за достъп до локален кеш на MP

P = стойност за брой пропуски в кеш паметта

N = броя на PIM възлите

и параметрите $\%W_{PIM}$, M и N са независими.

Полученият израз за M показва, че когато N > M, използването на PIM възел винаги ще бъде от полза и ще има по-добра производителност от система без PIM възел. Моделът дава също така основна информация за разпределяне на работата съгласно двата основни параметъра на PIM:

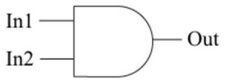
- Броя на PIM възлите;
- Частта от общата работа, която може да бъде подадена на PIM.

Въпреки че е трудно да се определят тези параметри за конкретно предназначение, разглеждайки ги в определен диапазон, може да получим приемлива представа за възможностите, които предлагат архитектури с различен брой PIM възли.

ГЛАВА III. Експериментална методология за симулация и хардуерно изпълнение на mPIM ядро

3.1 Симулация на mPIM ядро

За дизайна и структурирането на PIM ядро използваме SystemC, а симулацията извършваме чрез ModelSim. SystemC е комплект от C++ класове и макроси, които позволяват стъпково изпълнение на базата на събития. Това е описателен език за разработка на хардуер с цел симулиране и емулиране по начин, по който стандартният C++ език не позволява. Дизайнът се създава от модули, включени в SystemC библиотеките, които съдържат необходимите вход-изходи за деклариране на самите модули. Във всеки модул имаме конструктори, именувани по същия начин като модула. След като дефинираме модулите с техните конструктори, ние описваме действието на модула в SystemC конструктор. Това е частта от кода, която има поведението на реален хардуерен процес. Може да имаме различни нишки във всеки модул и всеки един от тях ще се изпълнява едновременно и паралелно. Изпълнението на много нишки едновременно е частта, която стандартният програмен език C/C++ не може да изпълни, тъй като там процесите се изпълняват последователно. Нишките могат да реагират на сигнали, на промяна в сигнала от тактовата честота, както и на фиксирано симулационно време.

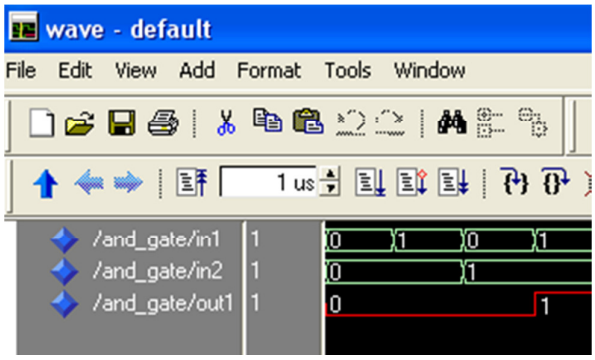


In1	In2	Out
0	0	0
0	1	0
1	0	0
1	1	1

Фиг. 3.1 Логически елемент "И" – "AND GATE"

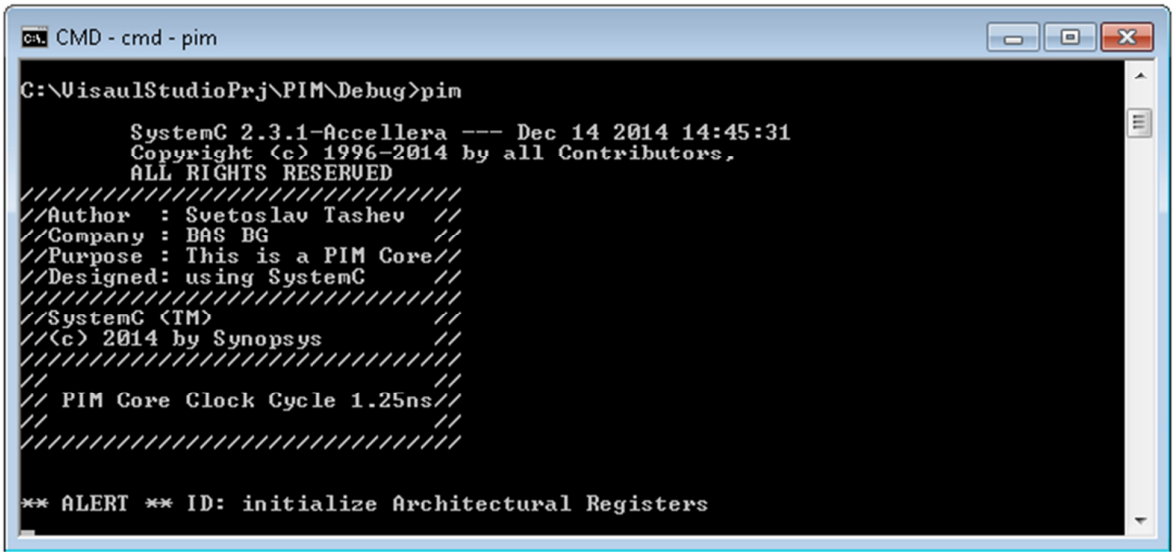
Тук е показан примерен дизайн на логически елемент "И" – "AND GATE" чрез ModelSim. Декларирането на този елемент чрез SystemC се извършва по следния начин:

```
#include "systemc.h"
SC_MODULE(and2)          // declare and2 sc_module
{
  sc_in<bool> In1, In2;      // input signal ports
  sc_out<bool> Out;          // output signal ports

  void do_and2()             // a C++ function
  {
    Out.write(In1.read() * In2.read());
  }

  SC_CTOR(and2)              // constructor for and2
  {
    SC_METHOD(do_and2); // register do_and2 with kernel
    sensitive << In1 << In2;   // sensitivity list
  }
};
```



Фиг. 3.2 Логически елемент "И" – "AND GATE"
Описание на SystemC и симулация чрез ModelSim

За дизайна на PIM сме избрали RISC архитектура поради множеството съществуващи разработки, както и лесното изпълнение и интеграция. Броят на елементите и топлоотделянето на този тип ядра са достатъчно малки, за да ни позволят интегриране в паметта. Нашите PIM ядра са настроени да поддържат тактова честота от 1.25 наносекунди. Това са 800 мегахерца, същата тактова честота, работеща върху HMC DRAM. Наличието на напълно функциониращо RISC ядро ни отваря възможностите за персонализация на спецификациите на процесора. Така процесорите може да са подходящи при изпълнението на всякакви приложения.

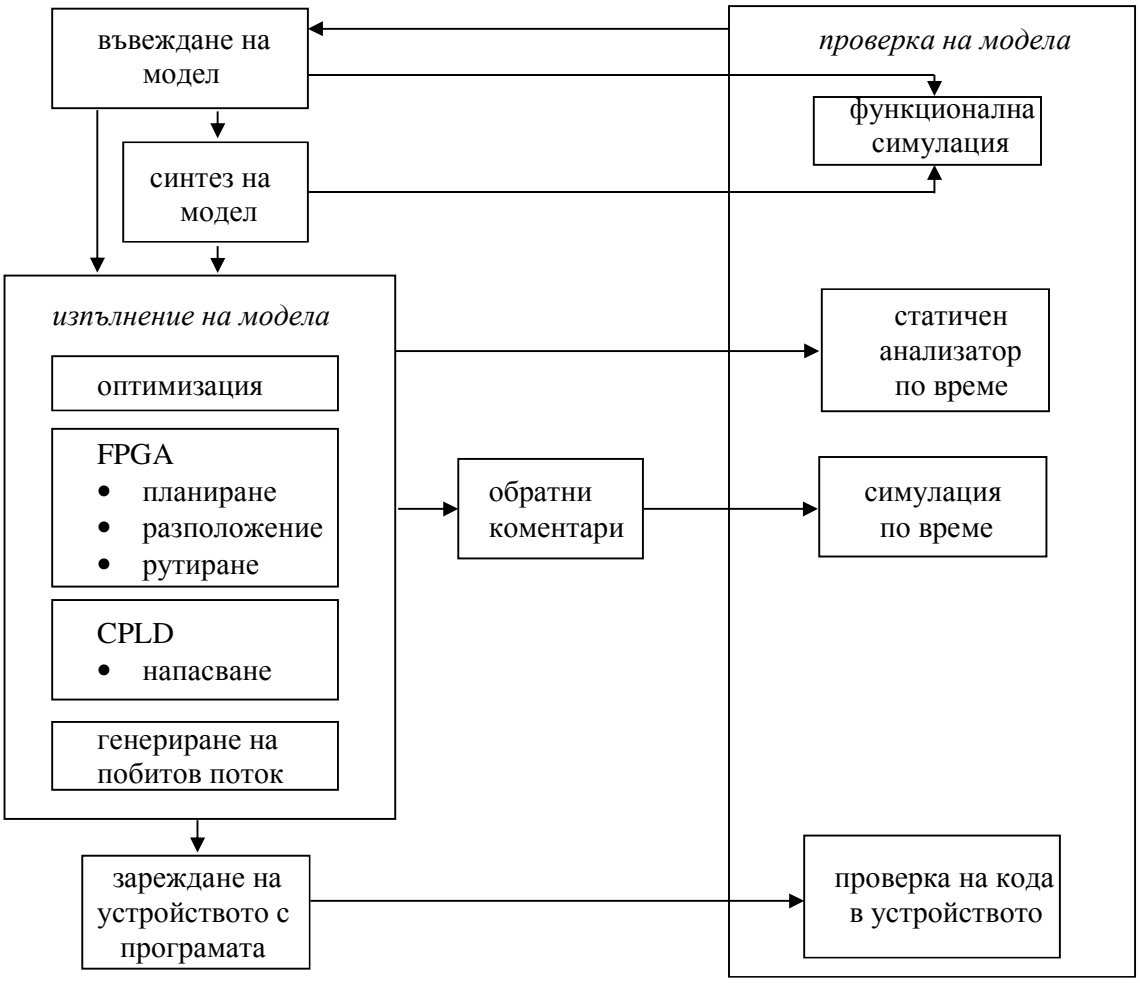


Фиг. 3.3 SystemC Model Debugging на PIM ядро

Описаната архитектура е съвместима и може да бъде синтезирана в платформа FPGA. Всички регистри, шини и вход-изходи са достъпни за модификации чрез препрограмиране на SystemC кода. На този етап имаме симулация на поведението от архитектурата на процесора. Поведенческата симулация позволява високо ниво на абстракция на ядрото. Функционалността на модулите може да бъде тествана без проверки за състезание на сигналите. Грешките, установени по време на симулациите, са лесно отстраними в началния стадий на дизайна.

3.2 Етапи на проектиране и хардуерно изпълнение на предложения модел PIM ядро

Изграждането на определен проект в средата ISE WEBPack е итеративен процес на симулация чрез проверка на логиката и проверка на поведението при състезание на сигналите, изпълняване и проверка на създаден модел до установяване на неговата коректност и финалното му завършване. Системата за разработка на Xilinx позволява бързи промени на споменатите стъпки по време на целия цикъл. Хардуерните устройства имат възможност за неограничен брой препрограмирания, без необходимост от физическо унищожаване при грешки в програмирания вече код.



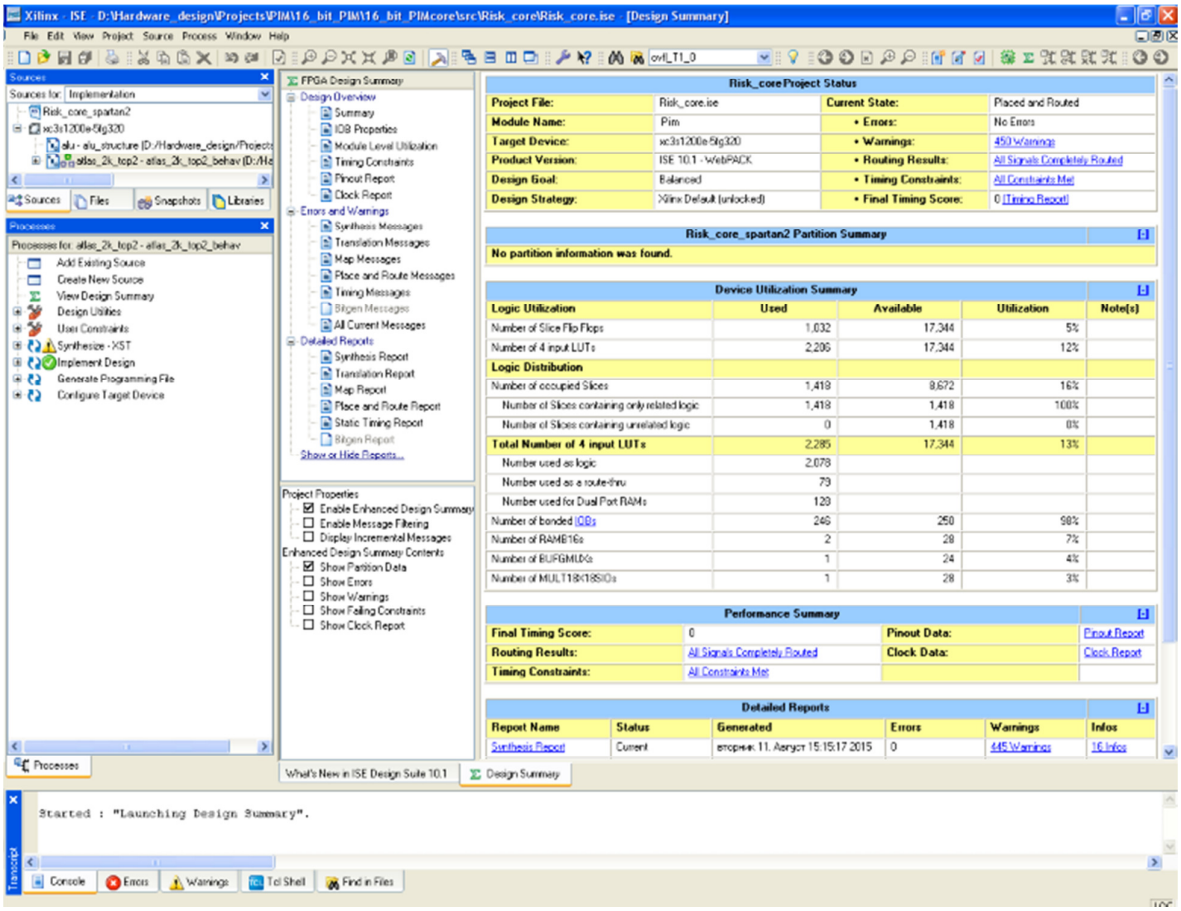
Фиг. 3.4 Блоксхема на проектиране

За реализацията на предложения модел се използват продукти на Xilinx, предназначени за FPGA.

3.3 Емулация и хардуерно изпълнение на предложения модел PIM ядро

За емулацията на предложения PIM модел използваме софтуерния пакет на Xilinx - ISE WEBPack (ISE – Integrated Synthesis Environment – или Среда за Интегриран Синтез). В този пакет е предоставен пълен набор от инструменти за програмиране на FPGA и CPLD, като се предлага HDL – Hardware Description Language, за синтез и симулация. Пакетът включва инструменти за програмиране и реализация на проекта, анализиране на времевите промени, промяна на настройките спрямо целта на устройството. ISE WEBPack поддържа цялата фамилия устройства на Xilinx, като се започне от FPGA серията от Virtex до Virtex5, FPGA серията от Spartan II до Spartan-3, както и CPLD серията CoolRunner. За реализация на ядрото, предвидено за PIM архитектурата, е избрана фамилията FPGA.

Основният потребителския интерфейс на ISE е навигаторът на проекта, който включва йерархия на проектиране (Workplace), редактор за програмния код (Workplace), изходна конзола (Transcript) и йерархия на процесите (Processes).



Фиг. 3.5 Графичен интерфейс на Xilinx ISE WEBPack

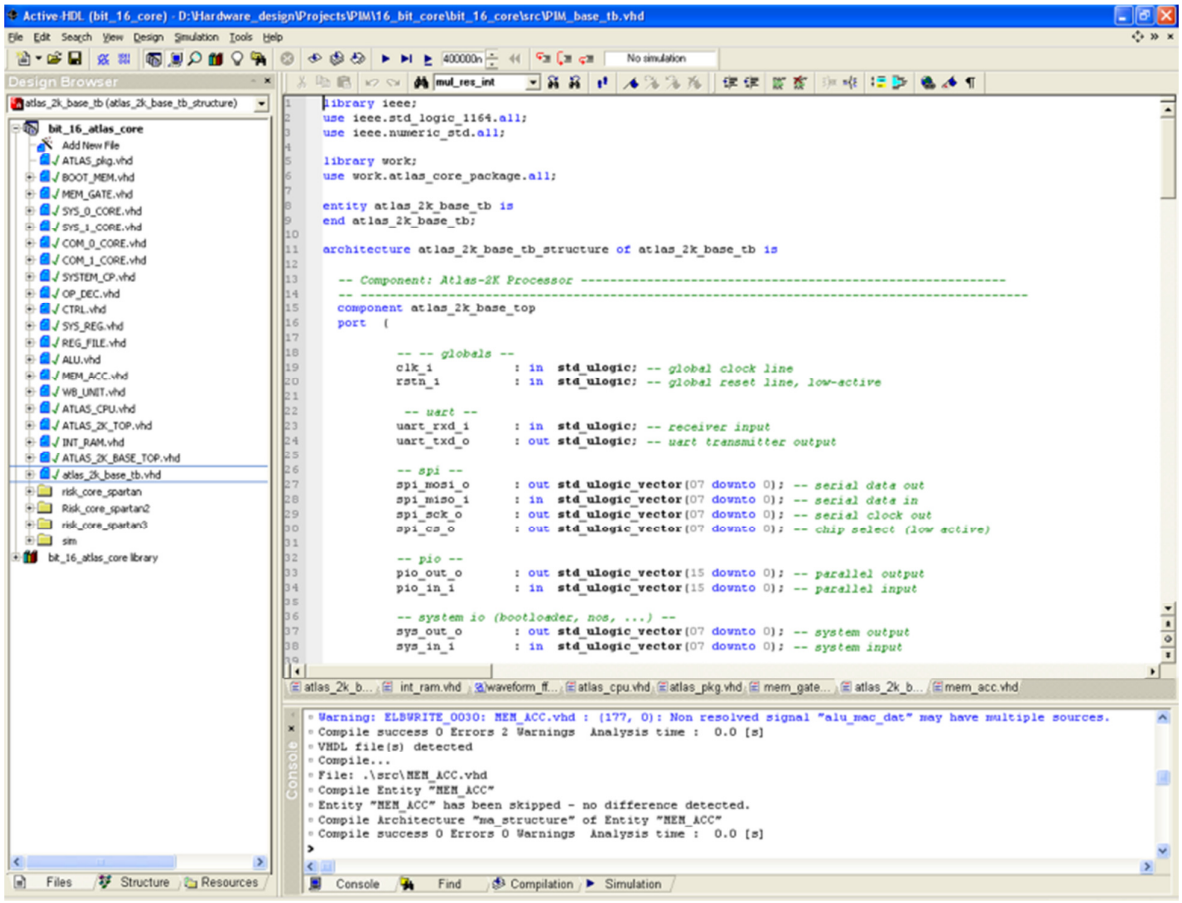
В своята същност ISE WEBPack представлява интегрирана среда за проектиране, изградена на модулен принцип, чиито зависимости се тълкуват от ISE и се показват като дървовидна структура. За едночипов дизайн това може да бъде един основен модул с други модули, включени към главния.

Design entry module – дава възможност за разработка на проекти върху HDL за описание на схеми от високо ниво, както и разработка на проекти с помощта на класическите схемни методи за развой. Тестване на системно ниво може да се извърши със симулатора ModelSim или чрез подобни HDL програми.

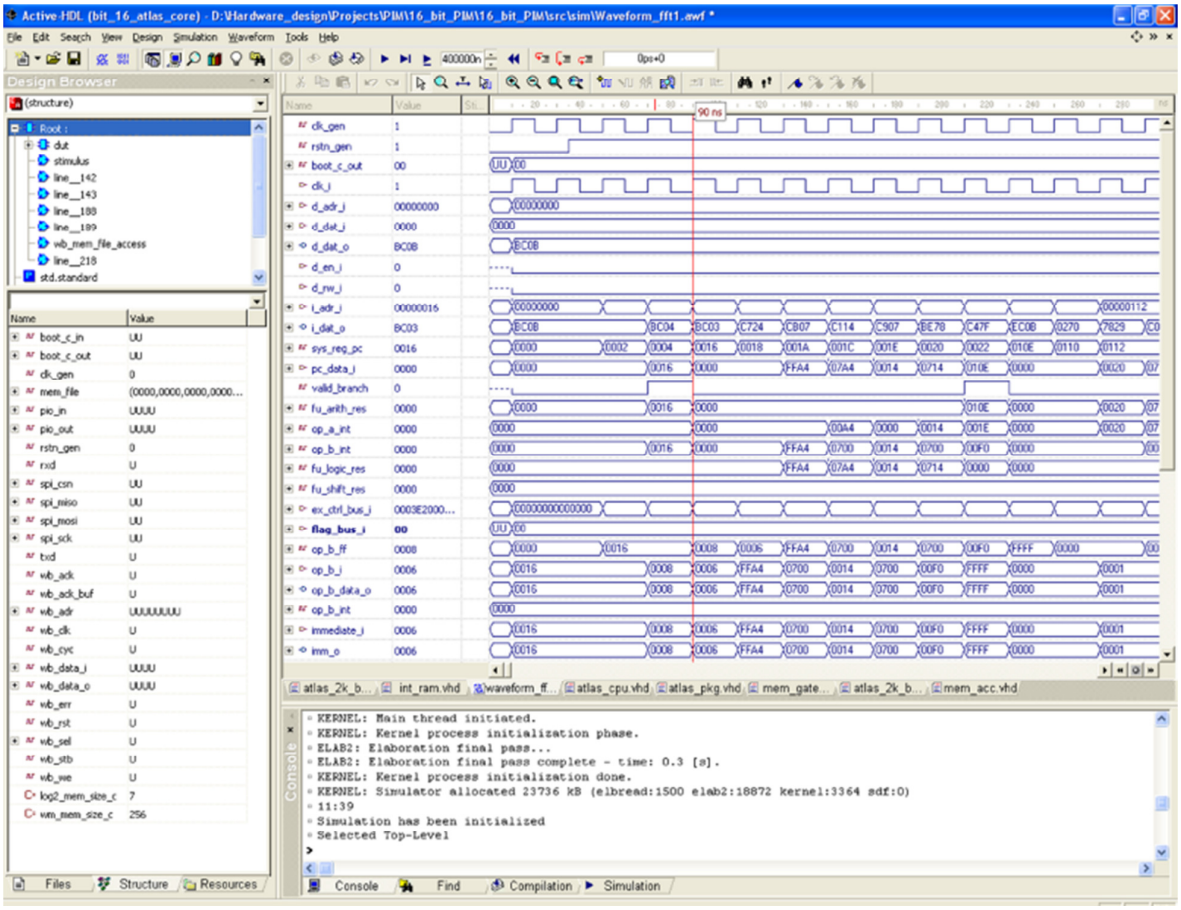
Fitter – дава възможност за прилагане на вече разработения дизайн в чип структурата.

Programming – модул за програмиране на чипа.

Допълнителният пакет BackPack към софтуерния пакет ISE WebPack внедрява допълнителни модули, които биха могли да се използват при необходимост. Това могат да бъдат пакети, които осигуряват условия за лесна проверка на функционалността при разработка на новия дизайн, както и да дават общ поглед върху физическите ресурси, с които проектантът разполага в конкретния случай. Примерни допълнителни пакети са ChipViewer, FPGA Editor, FloorPlaner, CoreGenerator, ModelSim XE Starter Edition и т.н.



Фиг. 3.6 Код и симулиране - 1, чрез ISE WEBPack



Фиг. 3.7 Код и симулиране - 2, чрез ISE WEBPack

При проектирането на PIM модула се реализират няколко от основните компоненти чрез HDL езика. Това са:

DPM Control – управление на двупортова памет;

I/O Control – проверка и управление на входно-изходните потоци;

Arithmetic CU – управление на аритметичното устройство;

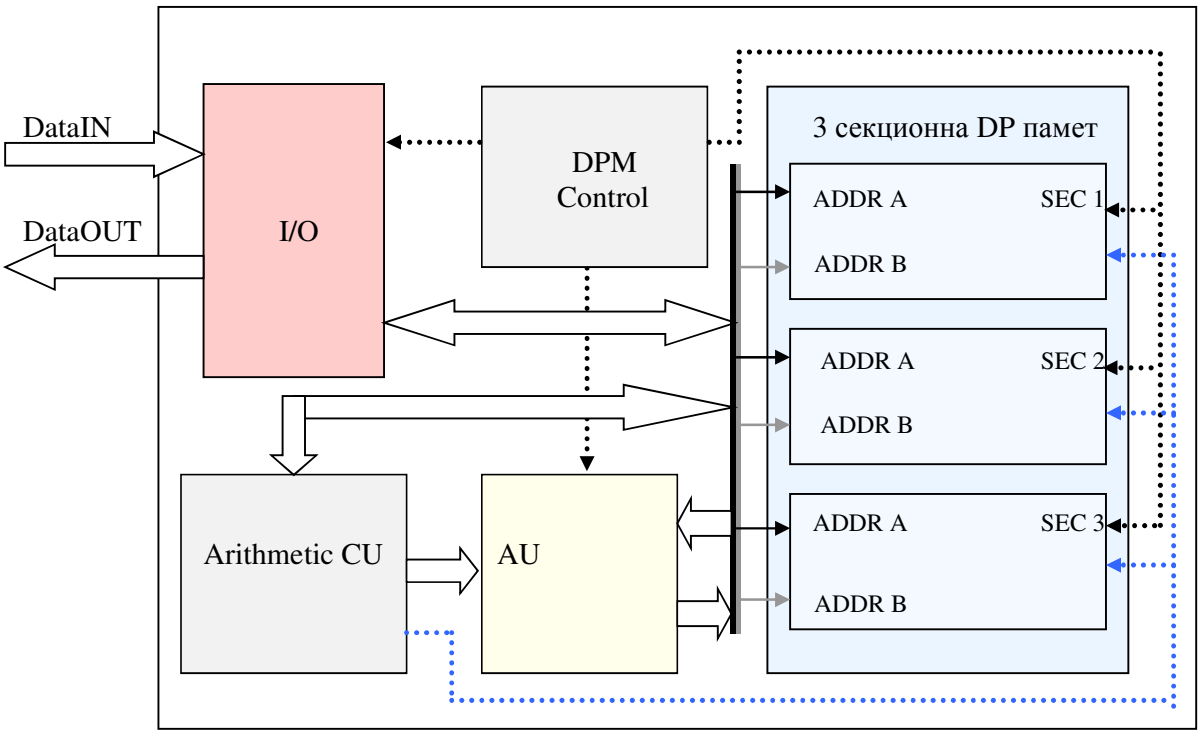
AU – структура на аритметично устройство на PIM модула.

Основната цел на емулацията е проверка на симулирания модел на PIM ядрото, което ще се ползва за симулация на цялостната система в следващия етап на проекта.

3.4 Реализация на различните PIM модули с HDL

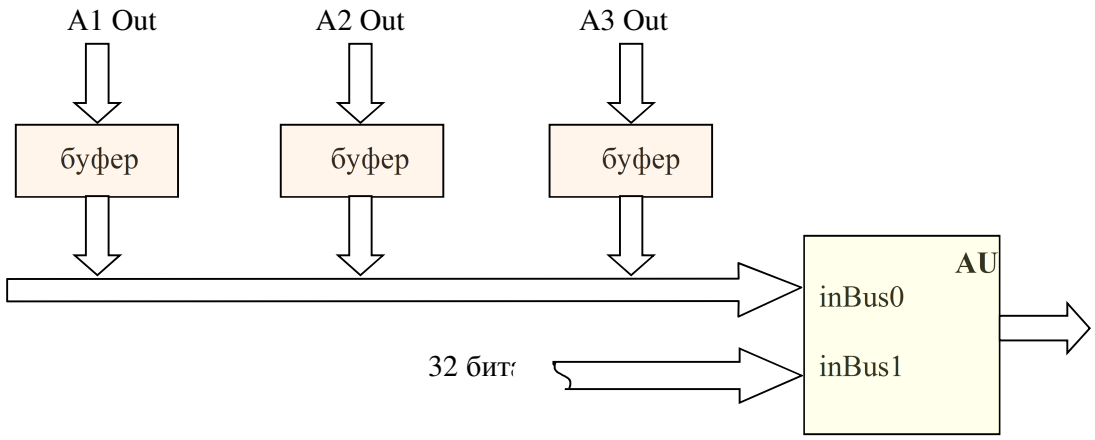
Различните компоненти на ядрото са изградени чрез HDL – Hardware Description Language. Както споменахме по-горе, един от тези компоненти е DPM Control. Компонентът се грижи за управление на паметта (запис/четене). От този модул се контролира вътрешният поток на информацията от и към аритметичното устройство AU, като в него влизат само данни от вътрешната памет. Външният входен поток

предварително се записва в трисекционната двупортова памет, управлявана от I/O Control модула и DPM Control. Хардуерната реализация е направена така, че на първия порт в 3-секционната памет се мултиплексират само адресите за четене и запис.



Фиг. 3.8 PIM модул

В първия порт може да постъпват както адреси за четене и запис, така и външен входно-изходен поток. Достъпът до AU се реализира по два 32 битови канала с 3 етапни буфера.



Фиг. 3.9 Достъп до AU

За по-голяма яснота на PIM архитектурата са представени опростени вход-изходи на основните компоненти. I/O Data Out също се реализира чрез буфери с 3 състояния. Аритметичното устройство извършва операциите по извличане и декодиране на инструкцията от 3-секционната памет в PIM, както и извличането на операнд и подаването му към DataOUT чрез I/O Control. DPM Control съгласува входно-изходните потоци към паметта на модула. Адресацията се извършва изцяло от AU. За тази цел е създаден AU модул – DPM adr, който се грижи изцяло за адресацията на 3-секционната памет. Достъпът до входните канали на AU е реализиран също така чрез 3 буфера с 3 състояния, като данните в него постъпват задължително от PIM паметта. DPM Control логиката е изградена така, че имаме възможност за управление едновременно на две от трите секции в PIM модула.

По-долу са представени вход-изходите на основния модул за управление на паметта:

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all;
USE ieee.std_logic_arith.all;
use AU_Pack_Xil.all;

entity CROMadr_minrsrc_gen is
      port (
            Pair_num:   in    unsigned(D1_max_local_pairs_exp -1 downto 0);
            en_shft:    in    std_logic;  -- enable shift
            init_shft:  in    std_logic;  -- initialize shifter
            CROM_adr:   out   unsigned(CR_sclr_pln_addr_width - 1 downto 0);
            CROM_adr_dld:     out unsigned(CR_sclr_pln_addr_width - 1 downto
0);
            clk :       in std_logic;
            rst :       in std_logic
      );
end CROMadr_minrsrc_gen;
```

За реализация на логиката четене и запис се използва модулът DPM Adr, който изглежда по следния начин:

```
entity  DPMadr_minrsrc_gen is
      port (
            Stg_Num:    in    unsigned(D1_stage_exp - 1 downto 0);
            Pair_num_r: in    unsigned(D1_max_local_pairs_exp -1 downto 0);
            Pair_num_w: in    unsigned(D1_max_local_pairs_exp -1 downto 0);
            DPMadr_0_r: out   unsigned(DM_pln_addr_width-1 downto 0);
            DPMadr_1_r: out   unsigned(DM_pln_addr_width-1 downto 0);
            DPMadr_0_w: out   unsigned(DM_pln_addr_width-1 downto 0);
            DPMadr_1_w: out   unsigned(DM_pln_addr_width-1 downto 0)
      );
end DPMadr_minrsrc_gen;
```

Двупортовата памет се дефинира по следния начин:

```
entity DPRAM_fxd_sclr_minrsrc is
      port (
            ADDRA:      in unsigned(DM_pln_addr_width-1 downto 0);
```

```
            ADDRB:      in unsigned(DM_pln_addr_width-1 downto 0);
            --ADDRA_w:  in unsigned(DM_pln_addr_width-1 downto 0);
            --ADDRB_w:  in unsigned(DM_pln_addr_width-1 downto 0);
            DIA :       in std_logic_vector(DPBusWidth - 1 downto 0);
            DIB :       in std_logic_vector(DPBusWidth - 1 downto 0);
            -- DIPA:    in std_logic_vector(1 downto 0);
            -- DIPB:    in std_logic_vector(1 downto 0);
            DOA :       out std_logic_vector(DPBusWidth - 1 downto 0);
            DOB :       out std_logic_vector(DPBusWidth - 1 downto 0);
            -- DOPA:    out std_logic_vector(1 downto 0);
            -- DOPB:    out std_logic_vector(1 downto 0);
            wen_A :     in std_logic;
            wen_B :     in std_logic;
            EN_A :      in std_logic; -- enable A port
            EN_B :      in std_logic; -- enable B port
            --wenb :    in std_logic;
            --oen :     in std_logic;
            --oenb :    in std_logic;
            CLK :       in std_logic;
            rst :       in std_logic
      );
end DPRAM_fxd_sclr_minrsrc;
```

Управлението на входно-изходния поток се описва по следния начин:

```
entity IO_fxd_sclr_minrsrc is
      port (
            xn_re:            IN std_logic_VECTOR(BusWidth -1 downto 0); --
Input data : Real component
            xn_im:            IN std_logic_VECTOR(BusWidth -1 downto 0); --
Input data . Imaginary component
            xk_re:            OUT std_logic_VECTOR(BusWidth -1 downto 0); --
Output data: Real component
            xk_im:            OUT std_logic_VECTOR(BusWidth -1 downto 0);
      -- Output data: Imaginary component
            xn_index:   OUT std_logic_VECTOR(D1_max_stages -1 downto 0); --
Index of input data.
            xk_index:   OUT std_logic_VECTOR(D1_max_stages -1 downto 0); --
Index of output data.
            Data_in:    out   std_logic_vector(DPBusWidth - 1 downto 0);  --
input for DPRAM port 0
            --Data_in_im: out       word;  -- input for DPRAM imaginary -
port 1
            IO_DP_WE:   out   std_logic;
            Data_out:in std_logic_vector(DPBusWidth - 1 downto 0);  -- output
from DPRAM real - port 0
            --Data_out_im:in        word;  -- output from DPRAM imaginary -
port
            IO_DPadr_in: out        unsigned(DM_pln_addr_width-1 downto 0);
            IO_DPadr_out: out       unsigned(DM_pln_addr_width-1 downto 0);
            inc_in_ixcntr:in  std_logic;
            EOLd:       out         std_logic;  -- end of load
            EOLd_dld:   out         std_logic;  -- end of load
            inc_out_ixcntr:in std_logic;
            EOULd:            out         std_logic;
            --EOULd_dld:      out         std_logic;
            --inc_in_ixcntr: in     STD_LOGIC;  -- increment input index cntr
            --inc_out_ixcntr:in     STD_LOGIC;  -- increment output index
cntr
            -- BR_out:        in          STD_LOGIC;  -- bit reverse output
            --Load_phase: in        STD_LOGIC;
            clk:        in    STD_LOGIC;  -- system clock
      rst:        in    STD_LOGIC  -- system reset , added  );
end IO_fxd_sclr_minrsrc;
```

ГЛАВА IV. Симулация на цялостни системи, ползващи изчислителни ресурси в паметта

4.1 Избор на симулативно натоварване

Основната част на дисертационния труд е представянето, описанието и проектирането на нови многопроцесорни компютърни архитектури с допълнителни изчислителни елементи в основната памет. Ние изучаваме цялостното подобрение на съществуващите конвенционални архитектури при използване на новопроектираната архитектура. Симулиращите модели са разработени на базата на съществуващи данни, като се сравняват времеви маркери чрез измерване на изходна производителност. Използване на допълнителни компютърни елементи в паметта ни позволява да направим подобрения в сегашните ограничения на конвенционалните архитектури.

Проектирането на хетерогенна мултипроцесорна и многоядрена архитектура с обща ISA е сериозно предизвикателство, като множество фактори трябва да бъдат взети под внимание. Аналитичният модел е създаден чрез симулации и резултатите са сравнени с данни от реално съществуващи компютърни системи за проверка на грешки в модела. За натоварването на различните избрани от нас системи използваме данни от няколко компании за оценка на ефективността на физическите сървъри. Доминиращи в индустрията са моделите на TPC-A, TPC-C, Netperf, Laddis, Kenbus, Sdet. Ако сравним отделните показатели на тези тестове, ние виждаме, че общите натоварвания имат прилики, когато машините не се използват за обща употреба. Отделните показатели, като процент от общото натоварване при неспецифични задачи между различните стандарти за разклонения, са:

TPC-A	16.9%
TPC-C	18.9%
Netperf	18.6%
Laddis	18.9%
Kenbus	16.3%
Sdet	17.8%

За общото натоварване на новосъздадената архитектура ще се спрем на информацията от стандарта TPC (Transaction Processing Performance Council). TPC е лидер на индустриалните стандарти, който се използва предимно за оценка на ефективността на физическите сървъри преди освобождаването им в операции. Причините да се спрем точно на този модел натоварвания са две: по-голяма част от новопроизведените сървъри се тестват с този модел натоварване; налични са данни от реално тествани

многопроцесорни машини, които може да ползваме за проверка на резултатите от симулативния модел. TPC Benchmark, On-Line Transaction Processing (OLTP) натоварва сървърите със следните инструкции:

STORES	12%
LOADS	25.2%
INTEGER OPERATIONS	42.1%
FLOATING-POINT	1.8%
BRANCHES	18.9%

Нашите симулационни модели са базирани на: система с Intel Xeon E7-2870 с две гнезда и два процесора, система с Intel Xeon E7-4870 с четири гнезда и четири процесора, система E7-8870 с осем гнезда и осем процесора. Тези компютърни системи са многопроцесорни и многоядрени. Отделните ядра между различните системи по същност и характеристики са идентични и имат същите индивидуални характеристики за ефективност. При реални тестове върху физически съществуващи сървъри, натоварени чрез модела на TPC OLTP, се вижда, че удвояването на процесорните ядра не удвоява резултата при изчисленията.

Процесор	# Брой ядра	TPC резултат	Резултат за ядро
E7-2870	20	1560.70	78.04
E7-4870	40	2862.61	71.57
E7-8870	80	4614.22	57.68

Фиг. 4.1 TPC резултат при увеличаване на ядрата

Резултатите са потвърдени от тестове над физическите сървъри и нашия симулационен модел. Ускорението между различните системи се изчислява чрез:

$$S = \frac{\text{Tsys2}}{\text{Tsys1}}$$

S – Цялостно ускорение на системата спрямо предходната
Tsys1 – Резултат преди подобрението – система 1
Tsys2 – Резултат след подобрението – система 2

Действителното измерено ускорение след удвояване на ядрата между E7-2870 и E7-4870 е 1.8. След удвояването на ядрата от E7-4870 към E7-8870, ускорението е едва 1.6. Това е очакван резултат, върху който влияят множество фактори. В основата е засилената конкуренция за споделените ресурси или по-горе споменатият проблем „Von Neumann bottleneck“. В допълнение към това, не всички сегменти могат да се изпълняват паралелно.

4.2 Опитна постановка на тестваните модели

За да се симулира правилно предложената цялостна структурна система и да получим реалното ускорение на нейната работоспособност, ще сравним получените резултатите спрямо съществуващите критерии на физически сървъри и архитектури. Симулацията се базира на моделиране на последователни заявки към процесорните ядра и използва вероятностно разпределение на задачите от избрания TPC тест. Условията за симулация на всички архитектури са такива, каквито са описани по-горе, плюс включване на времето на престой на конвенционалните процесори, причинени от кеш пропуска и грешно предвидени разклонения. При сравняване на две еднакви архитектури ние не трябва да взимаме под внимание пропуските от заявки към кеша, защото се предполага, че те ще са идентични. Добавяне на изчислителни елементи в паметта ще ни даде директен достъп до всички данни от основната памет и ние на практика няма да имаме кеш пропуски през PIM операциите. При оценка на работата на избраната архитектура с две гнезда, система с Intel Xeon E7-2870 процесори, използвайки Perfstat, имаме следните статистически данни:

500.2776842	време на теста в микросекунди
1077328	цикъла
3576240	обработени инструкции
755940	разклонения
52865	обръщения към кеша
9745	пропуски от кеша
770	грешки

Въпреки че броят на кеш пропуските изглежда незначителен – 0.181% от всички инструкции (или 18.434% от всички кеш заявки), санкциите при кеш пропуски са огромни. Като цяло заявките за четене от кеша са критични за работата на системата, тъй като те са необходими за напредъка на приложението. Ръководството за Intel Xeon процесори „The Performance Analysis Guide for Intel Xeon Processors“ [3] предоставя груб приблизителен анализ на изхабените цикли за достъп до следващото ниво, необходими след кеш пропуск:

L1 кеш достъп	~4 цикъла
L2 кеш достъп	~10 цикъла
L3 кеш достъп	~75 цикъла
RAM достъп	~100-300 цикъла (време за достъп към основната памет)

Общо изхабените цикли на изчакване на всяко изчислително ядро, причинени от пропуск при четене на кеша, могат да бъдат изчислени, като се вземат предвид необходимите цикли за достъп до следващото ниво:

CPU Time = (CPU exec clock cycles + memory stall cycles) * clock cycle time
Memory stalls = read stalls + write stalls
Read stall cycles = reads per program * read miss rate * read miss penalty

Като цяло пропуск от кеша ще ни коства около 100 цикъла на основното изчислително ядро. Въпреки годините на проучване и моделите, които се прилагат за предварително зареждане на страници в кеша, съвременните компютри не са в състояние да получат сто процента от заявките към кеша. При прилагането на изчислителни ядра в основната памет ние няма да бъдем изправени пред този проблем. PIM ядрата ще имат директен достъп до всички страници, заредени в основната памет.

Друг фактор, който трябва да отчетем, е забавянето, причинено от неправилно предвиждане на разклоненията. Има редица приносители на забавянето след грешно предвиден преход. Основният приносител е дължината на процесорния буфер. Ние ще трябва отново да презаредим буфера след разклонение, а това допълнително може да предизвика потенциален кеш пропуск. Различни проучвания показват, че санкциите от неправилно предвидени преходи варират от 10 до 35 процесорни цикъла, в зависимост от дължината на буфера. Реалните тестове над Intel Pentium Pro процесор ни дават средно 20 цикъла санкция след грешно предвиден преход.

4.3 Резултати

Разглежданите симулативни модели взимат предвид текущите физически ограничения и технологиите, достъпни за разработка. Максималната скорост на PIM ядрата има реално ограничение от скоростта на паметта, в която ще работят. Поради скоростта на паметта конвенционалните процесори използват коефициентен делител при работата с паметта за синхронизиране на тактовата честота. Често коефициент от 1/3 или 1/6 се използва за синхронизация на шасито с ядрото. Сравнено с конвенционалните процесори, PIM ядрата са значително по-бавни. Новоразработената архитектура ползва 1/3 делител за скоростта на основната памет към основните конвенционални ядра. Това е забавяне от 66% на PIM ядрата. Следват моделите и тяхното описание:

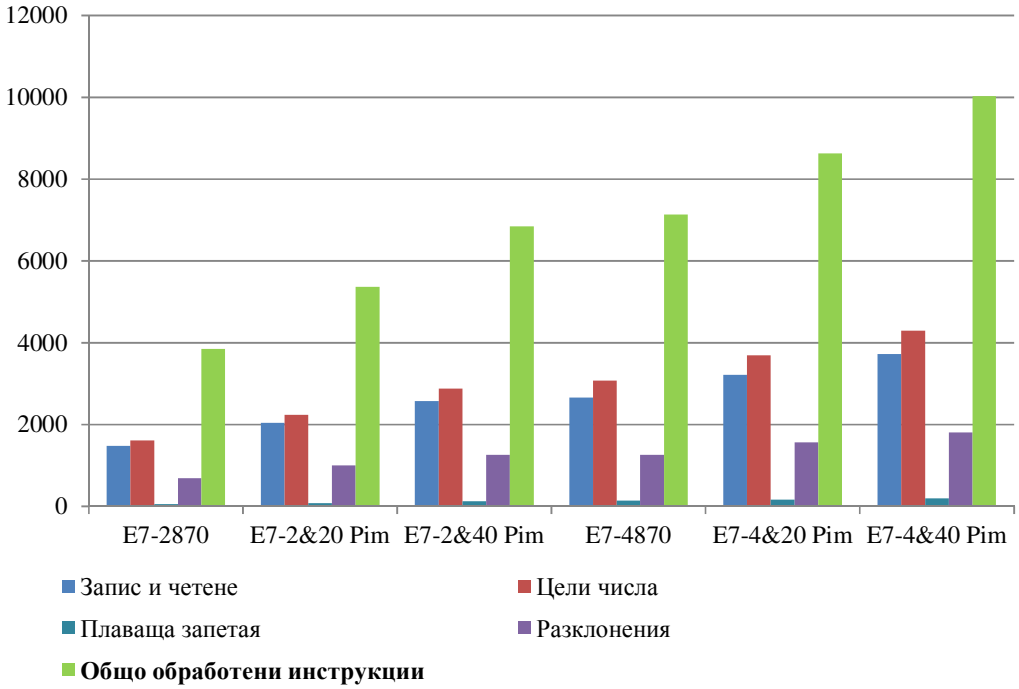
v31c2: E7-2870 симулативен модел на конвенционална система Intel Xeon E7-2870 с две гнезда и два процесора. Архитектура с 20 процесорни ядра.
v31c2P2: E7-2870 симулативен модел на конвенционална система Intel Xeon E7-2870 с две гнезда и два процесора, плюс 20 допълнителни PIM ядра.
v31c2P4: E7-2870 симулативен модел на конвенционална система Intel Xeon E7-2870 с две гнезда и два процесора, плюс 40 допълнителни PIM ядра.
v31c4: E7-4870 симулативен модел на конвенционална система Intel Xeon E7-4870 с четири гнезда и четири процесора. Архитектура с 40 процесорни ядра.
v31c4P2: E7-4870 симулативен модел на конвенционална система Intel Xeon E7-4870 с четири гнезда и четири процесора, плюс 20 допълнителни PIM ядра.
v31c4P4: E7-4870 симулативен модел на конвенционална система Intel Xeon E7-4870 с четири гнезда и четири процесора, плюс 40 допълнителни PIM ядра.

	v31c2	v31p2c2	v31p4	v31c4c2	v31p2c4	v31p4c4
Branch 10	353	353	353	346	346	346
Branch 20	342	342	342	308	308	308
Branch 30	NA	NA	NA	316	316	316
Branch 40	NA	NA	NA	289	289	289
Integer 10	791	791	791	757	757	757
Integer 20	823	823	823	770	770	770
Integer 30	NA	NA	NA	809	809	809
Integer 40	NA	NA	NA	737	737	737
FP 10	29	29	29	30	30	30
FP 20	34	34	34	36	36	36
FP 30	NA	NA	NA	36	36	36
FP 40	NA	NA	NA	38	38	38
LS 10	724	724	724	651	651	651
LS 20	754	754	754	704	704	704
LS 30	NA	NA	NA	659	659	659
LS 40	NA	NA	NA	648	648	648
Pim		p20	p40		p20	p40
Branch 50	NA	150	143	NA	149	142
Branch 60	NA	160	149	NA	157	132
Branch 70	NA	NA	125	NA	NA	134
Branch 80	NA	NA	150	NA	NA	143
Integer 50	NA	321	308	NA	319	303
Integer 60	NA	305	307	NA	302	301
Integer 70	NA	NA	370	NA	NA	357
Integer 80	NA	NA	286	NA	NA	265
FP 50	NA	10	11	NA	10	11
FP 60	NA	10	10	NA	10	11
FP 70	NA	NA	24	NA	NA	23
FP 80	NA	NA	17	NA	NA	17
LS 50	NA	297	287	NA	295	277
LS 60	NA	268	260	NA	261	262
LS 70	NA	NA	248	NA	NA	237
LS 80	NA	NA	304	NA	NA	285
	E7-2870	**E7-2&20**	**E7-2&40**	**E7-4870**	**E7-4&20**	**E7-4&40**
LS	1478	2043	2577	2662	3218	3723
Integer	1614	2240	2885	3073	3694	4299
FP	63	83	125	140	160	202
Branches	695	1005	1262	1259	1565	1810
Total	3850	5371	6849	7134	8637	10034
		1.3950649	1.7789610	1.8529870	2.2433766	2.6062337
SpeedUp:						
	E7-2870	E7-2&20	E7-2&40	E7-4870	E7-4&20	E7-4&40
LS	1.00	1.38	1.74	1.80	2.18	2.52
Integer	1.00	1.39	1.79	1.90	2.29	2.66
FP	1.00	1.32	1.98	2.22	2.54	3.21
Branches	1.00	1.45	1.82	1.81	2.25	2.60
Overall Speedup	**1.00**	**1.40**	**1.78**	**1.85**	**2.24**	**2.61**

Branch 10 – Общо разклонения към процесорни ядра 1 до 9
Integer 20 – Общо заявки за изчисления на цели числа към ядра 10 до 19
FP 30 – Floating Point - Общо заявки за изчисления с плаваща запетая към ядра 20 до 29
LS 40 – Load Store - Общо заявки за четене и запис към ядра 30 до 39

Симулативните модели v31c2 и v31c4 са разработени с цел вторична оценка на симулациите. Резултатите от симулацията са сравнение спрямо резултатите при тест на физически сървъри E7-2870 и E7-4870 при TPC OLTP натоварване.

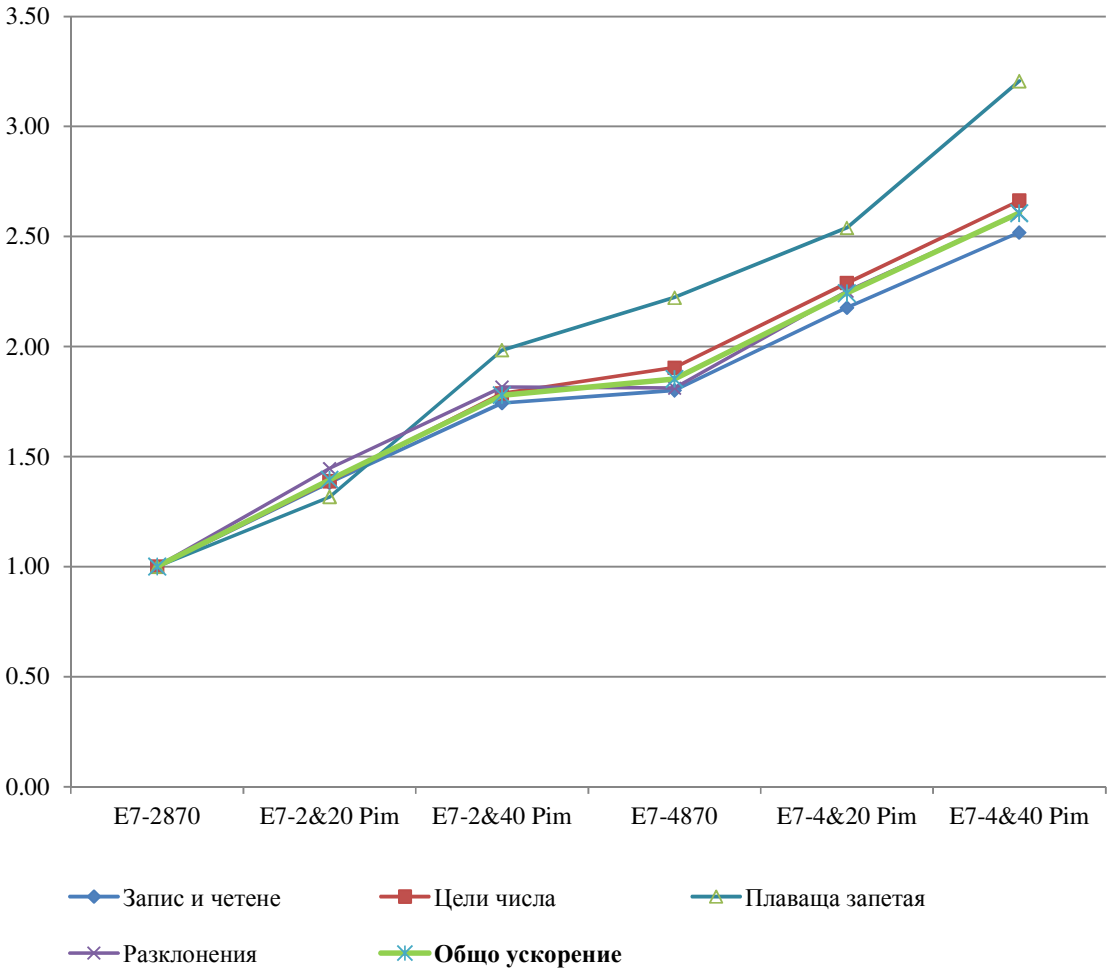
Общите комбинирани резултати за всички изчислени инструкции през времето на провеждане на серията тестове са следните:



v31c2: 3850 Общо обработени инструкции
v31c2P2: 5371 Общо обработени инструкции
v31c2P4: 6849 Общо обработени инструкции
v31c4: 7134 Общо обработени инструкции
v31c4P2: 8637 Общо обработени инструкции
v31c4P4: 10034 Общо обработени инструкции

Фиг. 4.2 Симулирани модели и резултати

Виждаме, че новосъздадената архитектура с двадесет ядра и допълнителни четиридесет PIM ядра в паметта ни дава почти идентичен резултат с конвенционална архитектура с четири гнезда и четиридесет процесора. Разликата ще бъде консумацията на енергия, тъй като предложената нова PIM архитектура ще бъде значително по-енергийно ефективна. Резултатите от ускорението показват, че системи с по-малко конвенционални процесорни ядра ще имат значително подобрение от помощните ресурси.



v31c2:	Сравнителна база на конвенционална система с 20 ядра
v31c2P2:	1.39 ускорение с 20 допълнителни PIM ядра
v31c2P4:	1.77 ускорение с 40 допълнителни PIM ядра
v31c4:	1.85 сравнителна база на конвенционална система с 40 ядра
v31c4P2:	2.24 ускорение с 40 конвенционални и 20 допълнителни PIM ядра
v31c4P4:	2.60 ускорение с 40 конвенционални и 40 допълнителни PIM ядра

Фиг. 4.3 Резултат на комбинираното ускорение

Сравнявайки ускорението между симулативните модели на двете конвенционални системи v31c2 и v31c4, респективно Intel Xeon E7-2870 и E7-4870, получаваме резултат от 1,852. TCP OLTP тестът над физическите сървъри със същата конфигурация е 1.834, което прави резултатите почти идентични и доказва нашия тест като валиден [12].

Заключение

Резултатите, постигнати чрез симулация и емулация на процесорни елементи в изчислителната памет, демонстрират убедително ефективността на предложената архитектура с PIM като възможност за търсене на алтернативни архитектурни решения за повишаване производителността на компютрите чрез решаване на проблема с облекчаване на информационния трафик между основните процесори и оперативната памет, улесняване при разработка на допълнителни ядра чрез намаляване на техните ресурси, подобряване на консумираната енергия от системата.

Научни и научно-приложни приноси

В резултат на изследванията, представени в настоящия дисертационен труд, са постигнати следните научни и научно-приложни резултати:

1. Предложени са многоядрени и многопроцесорни компютърни архитектури с допълнителни процесорни елементи, включени в основната памет.
2. Предложени са четири области за разпределяне на изчислителните функции в компютъра между основните конвенционални процесори и процесорните елементи в основната памет.
3. Предложена е промяна в основната диаграма на изпълнение на инструкциите и е създаден аналитичен модел на предложената архитектура.
4. Предложените архитектури са изследвани чрез симулационни процедури и са получени редица числови показатели, които доказват ефективността от използване на PIM за повишаване производителността на компютъра. Показана е адекватността на симулационните модели чрез сравнителни резултати от реални системи.
5. Емулирани са основни елементи от PIM възела чрез използване на най-съвременна FPGA технология, с което се потвърждава възможността за практическа реализация на предлаганите архитектури.

Насоки за бъдеща изследователска работа

1. Разработка на модел със споделено ядро за изчисления на инструкции с плаваща запетая между PIM ядрата.
2. Изследване на консумираната енергия и топлоотделянето на новоразработените системи.
3. Разработка на модел за използването на mPIM ядро за подобряване на пропускателната способност на шината на паметта – чрез компресиране на данните и изпращането им като отделни пакети.

Публикации по дисертационния труд

1. T. Tashev, **S. Tashev**, N. Tasheva. **Design of Advanced Computer Architectures, based on PIM – Processors in Memory**, Journal "Information Technologies and Control" (Year VIII, Nr. 3, 2010), ISSN: 1312-2622, 19-22.

2. **S. Tashev**, M. Marinova, V. Lazarov. **Multiprocessor Computer Architecture with Additional Computing Cores in the Memory**. Journal "Computer&Communications Engineering", ISSN 1314-2291, vol. 9, 1/2015, pp 49-54.

3. **S. Tashev**, V.Lazarov, T. Tashev. **Development of Processing Elements Inside the Memory to Aid Multiprocessor Computer Architectures**. Proceedings of the Fifth International Conference "Education, Science, Innovations" (ESI'13), European Politechnical University, ISSN 1314-5711, Pernik, June 9-10, 2015, под печат.

4. S. Tashev, Ph. Philipov, T. Tashev. **Emulation and Analytical Model of PIM Supplemental Computing Element via HDL.** Journal "Information Technologies and Control", ISSN: 1312-2622, под печат.

5. З. Златев, В. Лазаров, М. Иванова, Ф. Филипов, Н. Ташева, Ю. Зидарова, **С. Ташев**, Т. Ташев ***„Архитектура на видеодекодер на базата на FPGA"***, Международна конференция Автоматика и Информатика '02, София, 5 – 6 Ноември 2002, pp. 5 - 9

Използвана литература

1. Prof. Onur Mutlu, Carnegie Mellon, Computer Architecture 2015

2. Intel, Metrics - Branch Mispredict node 529600 2015

3. Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ Processor 2011

4. Architectures, ASPLOS 2009

5. Dr Mike Murphy, Coastal Carolina University, Computer Science 2011

6. HP Labs, Disaggregated Memory for Expansion and Sharing 2009

7. HP.com, Darrel D. Donaldson, AlphaServer 4100 Performance Characterization 2006

8. HMC Gen2 LOT Rev. 1.1 2014

9. Heterogeneous Chip Multiprocessors, 2005 vol.38 no.11

10. Prof. Yale N. Patt, The University of Texas at Austin, Computer Systems

11. Prof. Yale Patt, Accelerating Critical Section Execution with Asymmetric Multi-Core

12. TCP Benchmark Transaction Processing OLTP 2014

13. John L. Hennessy, Computer Architecture - A Quantitative Approach (4th edition), Morgan Kaufmann, 2006

14. John P. Shen, Modern Processor Design: Fundamentals of Superscalar Processors, Electrical and Computer Engineering, McGraw-Hill Science, 2004

15. Richard Gerber, The Software Optimization Cookbook (2nd edition), Intel Press, 2006

16. Christian Piguet, Low-Power Processors and Systems on Chip, CRC, 2005

17. Linda Null, The Essentials of Computer Organization and Architecture (2nd edition), Jones & Bartlett Pub, 2006

18. David Judd, Katherine Yelick, Exploiting On-Chip Memory Bandwidth, pages 122–134, Cambridge, Massachusetts, 2000

19. VIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP), 2003

20. D. Abts, S. Scott, D.J. Lilja, Verifying memory coherence in IPDPS, 2003

21. IBM Corporation, PowerPC Microprocessor Family: The Programming Environments for 32Bit Microprocessor, 2000

22. IBM Corporation, PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology, 2005

23. Murphy, R. and P. M. Kogge, The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems, Intelligent Memory Systems Workshop, ASPLOS-IX 2000, Boston, 2000

24. E. Im and K.A. Yelick, Optimizing sparse matrix-vector multiplication for register reuse, Proc. Intl. Conf. on Computational Science, 2001.

25. B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang., Imagine: Media Processing with Streams, IEEE Micro, April 2001

26. Sterling T., Brockman, J., Analysis and Modeling of Advanced PIM Architecture Design Tradeoffs, Proceedings of the ACM/IEEE SC2004 Conference, October 2004

27. Intel Corporation, Intel Itanium Architecture Software Developer's Manual, 2006

28. William J. Dally, James Balfour, David Black-Schaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. IEEE Computer, July 2008

29. William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. Queue 2(1):52–62, 2004

30. Josh A. Fisher, Paolo Faraboschi, Cliff Young, Embedded Computing: A VLIW approach to Architecture, Compilers and Tools. Morgan Kaufmann Publishing, 2005

31. H. Richard Kendall, Vincent W. Freeh, Paul W. Schermerhorn, Robert J. Minerick, Peter W. Rijks, Streaming extensibility in the modify-on-access file system, Journal of Systems and Software vol. 60, 2002

32. M.J. Absar, P. Marchal and F. Catthoor, Data Access Optimization of Embedded Systems through Selective Inlining Transformation, Proceedings ESTMED, Sept 2005

33. M.J. Absar, F. Catthoor, Analysis of Scratch Pad and Data Cache performance Using Statistical Methods, in Proceedings ASPDAC'06, Jan 2006

34. J. Darper et al., The Architecture of DIVA Processing-in-Memory Chip, The 16th ACM International Conference on Supercomputing, pp. 14-25, June 2005

35. Z. Ahmad, Cooperative Intelligent Memory, PhD thesis, University of Hertfordshire, April 2007

36. Tejas S. Karkhanis, James E. Smith, Automated design of application specific superscalar processors: an analytical approach, San Diego, 2007

37. Satnam Singh, Integrating FPGAs in high-performance computing: programming models for parallel systems -- the programmer's perspective, Proceedings of the 2007 ACM/SIGDA 15th international symposium on FPGA, 2007

38. Y. Kang et al., FlexRAM: Towards an intelligent memory system, ICCD, Oct 1999

39. K. Mai et al., Smart memories: A modular reconfigurable architecture, ISCA, June 2000

40. R. Sotudeh, Z. Ahmad, F. Bensaali, Intelligent Co-operative Processor in Memory Architectures, The Mediterranean Journal of Electronics and Communication, Vol. 3, 2007, pp 17-30 Germainscaling impaired applications. In Parallel Processing Workshops, 2009

41. Norm Jouppi, The Future Evolution of High-Performance Microprocessors, IEEE/ACM International Symposium on Microarchitecture, 2005

42. Jason E. Miller, Anant Agarwal, Software-based instruction caching for embedded processors, Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006

43. Roberto D. Kent, High Performance Computing Systems and Applications (1st edition), Springer, 2003

44. Christoforos Kozyrakis, "Scalable Vector Media-processors for Embedded Systems", UCB/CSD-02-1183, 2002

45. Brian R. Gaeke,"Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines", Proceedings of the International Parallel and Distributed Processing Symposium, Ft. Lauderdale, FL. April, 2002

46. J. Gebis, S. William, C. Kozyrakis, D. Patterson „VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM", 41st Design Automation Student Design Contest, San Diego, June 2004

47. Sumit Mediratta, Jeffrey Draper "Achieving On-chip Fault-tolerance Utilizing BIST Resources", WSEAS Transactions on Circuits and Systems, Issue 12, Volume 5, December 2006, pp. 1726 - 1733

48. Sumit Mediratta, Jeffrey Draper „Effective Realization of On-chip Fault-tolerance Utilizing BIST Resources", Proceedings of the 5th WSEAS Int. Conf. on Circuits, Systems, Electronics, Control & Signal Processing, November 2006

49. Hesham El-Rewini, Mostafa Abd-El-Barr "Advanced computer architecture and parallel processing" Wiley Interscience, 2005

50. Jon Stokes "Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture", San Francisco, 2007

51. Sivarama Dandamudi "Guide to RISC Processors: for Programmers and Engineers" Springer, 2005

52. Paolo Ienne, Rainer Leupers "Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon)", Morgan Kaufmann, 2006

53. Mostafa Abd-El-Barr, Hesham El-Rewini "Fundamentals of Computer Organization and Architecture" Wiley Interscience, 2005

54. Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, Pierre America “A general model of software architecture design derived from five industrial approaches”, Journal of Systems and Software vol. 80, 2007
55. Shyamkumar Thoziyoor, Jay Brockman, Daniel Rinzler “PIM lite: a multithreaded processor-in-memory prototype”, Proceedings of the 15th ACM Great Lakes symposium on VLSI, 2005
56. Bita Gorjiara, Daniel Gajski “FPGA-friendly code compression for horizontal microcoded custom IPs”, Proceedings of the 2007 ACM/SIGDA 15th international symposium on FPGA, 2007
57. Jong-Eun Lee, Kiyoung Choi, Nikil D. Dutt “Instruction set synthesis with efficient instruction encoding for configurable processors” ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 12, 2007
58. Ioannis Panagopoulos, Christos Pavlatos, George Papakonstantinou “A hardware extension of the RISC microprocessor for Attribute Grammar evaluation” Proceedings of the 2004 ACM symposium on Applied computing, 2004
59. Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, Dimitrios S. Nikolopoulos “Online power-performance adaptation of multithreaded programs using hardware event-based prediction” Proceedings of the 20th annual international conference on Supercomputing, 2006
60. Dongkeun Kim, Steve Shih-wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, John P. Shen „Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors” Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2004
61. Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, John P. Shen „Helper threads via virtual multithreading on an experimental itanium® 2 processor-based platform” Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, 2004
62. Mikhail Popovich, Eby G. Friedman, Michael Sotman, Avinoam Kolodny, Radu M. Secareanu “Maximum effective distance of on-chip decoupling capacitors in power distribution grids” Proceedings of the 16th ACM Great Lakes symposium on VLSI, 2006
63. Tobias Bjerregaard, Shankar Mahadevan “A survey of research and practices of Network-on-chip” ACM Computing Surveys (CSUR), 2006

64. Krishna Sekar, Kanishka Lahiri, Sujit Dey „Dynamic Platform Management for Configurable Platform-Based System-on-Chips" Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design, 2003
65. Ward Douglas Maurer "The effect of the Harvard architecture on the teaching of assembly language" Journal of Computing Sciences in Colleges, vol. 20, issue 5, 2005
66. Tay-Jyi Lin, Chie-Min Chao, Chia-Hsien Liu, Pi-Chen Hsiao, Shin-Kai Chen, Li-Chun Lin, Chih-Wei Liu, Chein-Wei Jen "A unified processor architecture for RISC & VLIW DSP" Proceedings of the 15th ACM Great Lakes symposium on VLSI, 2005
67. Shorin Kyo, Shin'ichiro Okazaki, Tamio Arai "An Integrated Memory Array Processor Architecture for Embedded Image Recognition Systems" Proceedings of the 32nd annual international symposium on Computer Architecture, 2005
68. Gianfranco Bilardi „Models for parallel and hierarchical computation" Proceedings of the 4th international conference on Computing frontiers, 2007
69. Robert S. Germain, Blake Fitch, Aleksandr Rayshubskiy, Maria Eleftheriou, Michael C. Pitman, Frank Suits, Mark Giampapa, T.J. Christopher Ward "Blue matter on blue gene/L: massively parallel computation for biomolecular simulation" Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2005
70. Oliverio J. Santana, Ayose Falcón, Alex Ramirez, Mateo Valero "Branch predictor guided instruction decoding" Proceedings of the 15th international conference on Parallel architectures and compilation techniques, 2006
71. Jian Li, José F. Martínez "Power-performance considerations of parallel computing on chip multiprocessors" ACM Transactions on Architecture and Code Optimization (TACO), 2005
72. José Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, Kei Davis "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications" Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 2006
73. Damon Mosk-Aoyama, Devavrat Shah "Computing separable functions via gossip" Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, 2006
74. D. Rolan, B.B. Fraguela, and R. Doallo. Adaptive set-granular cooperative caching. In High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on, pages 1 –12, feb. 2012
75. Elberfeld, M. and Textor, J. (2009). Ecient Algorithm for String-Based Negative Selection. In Andrews et al., pages 109-121., 2009

76. Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20. HP Laboratories Palo Alto, 2008
77. Yuejian Xie and Gabriel H. Loh. Scalable shared-cache management by containing thrashing workloads. In Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers, HiPEAC'10, pages 262–276, Berlin, Heidelberg, 2010
78. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fern´andez, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors. In Proceedings of the 1st conference on Computing frontiers , CF'04, pages 433–443, 2004
79. Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting interthread cache contention on a chip multi-processor architecture. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA, pages 340–351, 2005
80. Blas A. Cuesta, Alberto Ros, Mar´ıa E. G´omez, Antonio Robles, and Jos´e F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In Proceedings of the 38th annual international symposium on Computer architecture, ISCA '11, pages 93–104, ACM 2011
81. Jaekyu Lee and Hyesoon Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on, pages 1 –12, 2012
82. Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A full system simulator for x86 CPUs. In Proceedings of Design Automation Conference, DAC '11, 2011
83. Texas Instruments, TMS320C54x DSP CPU and Peripherals Reference Set, volume 1 edition, 2001
84. M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In Proceedings of the conference on Design, automation and test in Europe Volume 2. IEEE Computer Society, 2004
85. http://www.xilinx.com
86. http://www.amd.com
87. http://www.intel.com
88. http://www.cray.com
89. http://en.wikipedia.org/wiki
90. http://www.ibm.com
91. http://www.llnl.gov/
92. http://www.springerlink.com/home/main.mpx
93. http://www.verilog.com/
94. http://www.systemc.org/

95. http://lpsolve.sourceforge.ne

96. http://www.arm.com/products/CPUs/families/ARM9Family.html ARM9E Family of Embedded Processors

97. http://pages.cs.wisc.edu/wwd/rev4.pdf

98. http://www.broadcom.com/collateral/pb/2702-PB02-R.pdf BCM2702: High Performance Mobile Multimedida Processor

99. http://lpsolve.sourceforge.net/5.5/

100. http://www.iestcfa.org/panel_discussions.htm 1st IEEE Inter. Symposium on Industrial Embedded Systems, 2006

101. http://dspvillage.ti.com/docs/dsppproducthome.html Texas Instruments, TMS320C54x DSP CPU 2001

102. http://www.alphaprocessors.com/

103. http://www.nd.edu/~pim/

104. http://www.top500.org

105. http://www.sgi.com/

106. http://www.research.ibm.com/bluegene

107. http://www.computing.net/

108. http://www.apple.com

109. http://www.isc2004.org

110. http://www.sun.com

111. http://www.eurogrid.org

112. http://www.see-grid.org

113. http://www.nasa.gov

114. http://www.giss.nasa.gov/

115. http://www.microprocessor.sscc.ru/

116. http://www.spec.org

117. http://www.ieee.org/portal/site

118. http://www.aaai.org/home.html

119. http://cvpr.cv.ri.cmu.edu/

120. http://www.tomshardware.com/

121. http://www.onsemi.com/

122. http://www-03.ibm.com/servers/

123. http://www.asus.com/

124. http://www.nvidia.com/page/home.html

125. http://www.memory.com/

126. http://www.almaden.ibm.com/
127. http://www.isi.edu/~draper/papers/iscas06_pim2.pdf
128. http://www.watson.ibm.com/index.shtml
129. http://www.ai.mit.edu/projects/aries/course/notes/pim.html
130. http://www.par.univie.ac.at/research/report/node24.html
131. http://www-unix.mcs.anl.gov/mpi/
132. http://www.altera.com/
133. http://www.annapmicro.com/
134. http://www.tyanev.com/resources/books/ComputerOrganization/
135. http://www.infinibandta.org/home
136. http://www.cisco.com/
137. http://www.mellanox.com/
138. http://www.newscientist.com/home.ns
139. http://supercomputing.org/
140. http://www.atmel.com