Българска академия на науките

Институт по информационни и комуникационни технологии

ас. инж. Светослав Марианов Ташев

Многопроцесорни архитектури с изчислителни ресурси в основната памет

А В Т О Р Е Ф Е Р А Т

НА ДИСЕРТАЦИЯ
За присъждане на образователна и научна степен „Доктор“

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

Дисертационният труд е обсъден и насрочен за защита на заседание на секция „Научни пресмятания“ - ИИКТ за провеждане на предзащита, разширено с хабилитирани учени в Българска Академия на Науките, състояло се на 01.12.2015 г.

Дисертационният труд е представен на 111 страници, в които 2 таблици, 61 фигури, списък на литературните източници с 43 заглавия и списък на публикациите към дисертацията.

Защитата на дисертацията ще се състои на 25.02.2016 г. от 14:00 часа в зала 218 на блок 25А на БАН на открито заседание на научно жури в състав:

1. Проф. дмн Иван Димов
2. Проф. дтн Людмил Даковски
3. Проф. дтн Живко Железов
4. Проф. д-р Владимир Лазаров
5. Доц. д-р Алексей Егоров

Материалите по защитата са на разположение на интересуващите се в стая 215 на ИИКТ-БАН, ул. „Акад. Г. Бончев“, бл. 25А

Автор: ас. инж. Светослав Марианов Ташев

Заглавие: Многопроцесорни архитектури с изчислителни ресурси в основната памет

Обща характеристика на дисертационният труд

Актуалност на темата и обзор на основните резултати в областта

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. 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 шина между основната памет и процесора. Поставянето на изчислителни ресурси в паметта ни позволява да обработваме данни директно върху паметта, което облекчава информационния трафик между основните процесори и оперативната памет. Допълнителните изчислителни ресурси имат голямо предимство и при приложения, изискващи едновременна обработка върху множество от данни. Това са приложения работещи върху база от данни, обработка на различни сегменти от едно приложение, както и изпълнението на помощни потоци за подпомагане на основното приложение.

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 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;
- Проектиране на основните възли и елементи от предложените структури с използване на съвременни технологии за хардуерно проектиране.
- 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;

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.

Методология на изследването

В дисертационния труд е разработена и изследвана компютърна архитектура с допълнителни изчислителни модули в паметта – mPIM (Multiple Processing in Memory Modules). Последователно са разработени:

- Аналитичен модел на използването на изчислителни ресурси в паметта;
- Симулация на mPIM ядро, чрез помощта на архитектурен симулатор SUNSiman;
- Емулация и хардуерно изпълнение на предложения модел PIM ядро, както и хардуерна имплементация чрез софтуерния пакет WEBPack ISE на Xilinx.

Изследването и симулация на методите за ползване на mPIM са съсредоточени върху архитектури, съставени от основен конвенционален процесор или няколко процесора и много хомогенни малки процесорни ядра, разположени в паметта. Това са силносвързани архитектури, ползващи обща шина и памет за комуникация. Наличието на основен конвенционален процесор ни позволява да ползваме всички текущи разработки върху паралелизиране на програмите. Фокусът на предложенията е постигането на подобрения, без да се налагат допълнителни промени на програмите или тяхното прекомпилиране, като това остава скрито за програмиста. Основният процесор ще ни позволи също така да възлагаме различните сегменти върху ядрото, което ще ги изпълни най-ефективно.

Апробация на резултатите

Основните резултати от дисертацията са изложени в пет публикации, две от които са отпечатани в сборници на международни научни конференции, а три – в научни списания. Всички публикации са изцяло на английски език.

Списък на публикациите по дисертацията

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

Съдържание на дисертацията

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

1.1. Взаимодействие между процесора и основната памет

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 [1].

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 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

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.

1.2 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 хардуерсофтуер, който ги изпълнява. В зависимост от това къде е поставен интерфейсът се определя натоварването на компилатора или хардуера.

Фундаменталните разлики между различните видове системни 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.

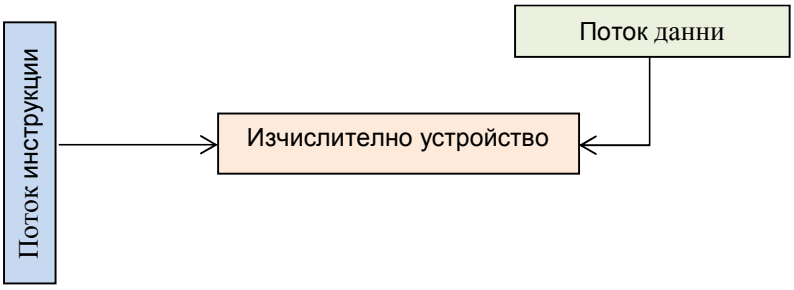
Организацията за изпълнението на системните инструкции от хардуера се определя от компютърната микроархитектура. Архитектурата на системните инструкции и микроархитектурата са тясно свързани. Архитектурата на системните инструкции представя програмния модел, като асемблерен език, докато микроархитектурата 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.

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

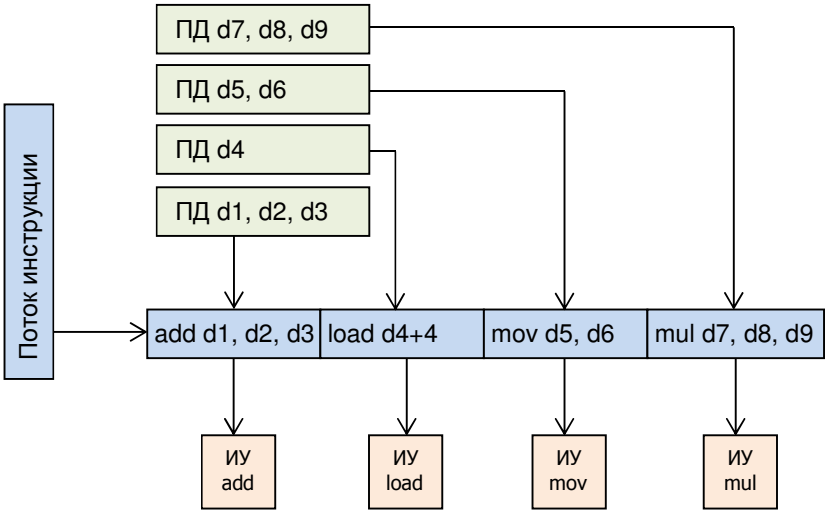


Фиг. 1.2 Единична инструкция оперира върху един елемент данни

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.



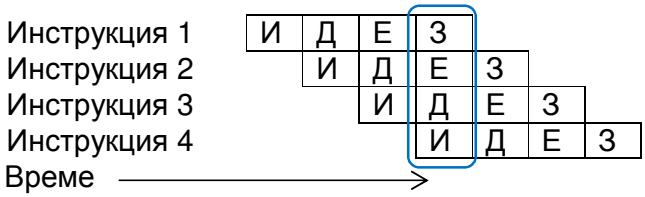
Фиг. 1.3 Архитектурата VLIW

*ПД – Поток данни

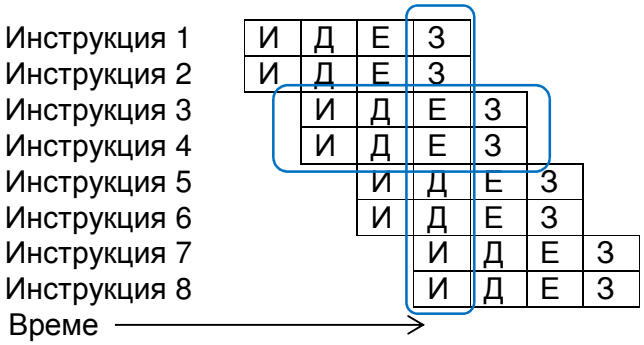
Различните функционални устройства, като аритметично-логическо устройство и математически копроцесор, могат да работят паралелно. Инструкциите от регистъра се изпълняват едновременно в заключена стъпка. Недостатък при този тип архитектура е сложността на компилатора. Компилаторът трябва да открие независимите 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.

В този вид компютърни архитектури попадат конвейерната и суперскаларната архитектура. Въпреки че конвейерната архитектура и суперскаларните архитектури са различни технологии, суперскаларните процесори също са и с конвейерна обработка на инструкциите [2]. 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



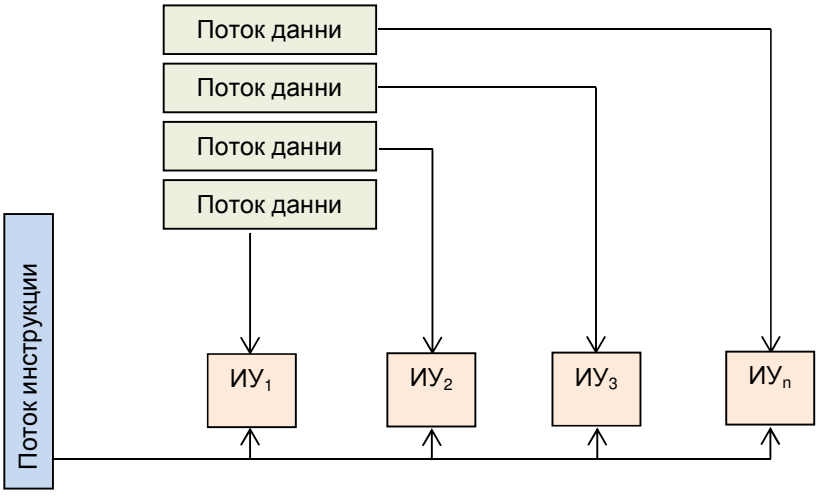
Конвейерно суперскаларно изпълнение на различните етапи от различни инструкции



Фиг. 1.4 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.

1.3.2 SIMD multiprocessing

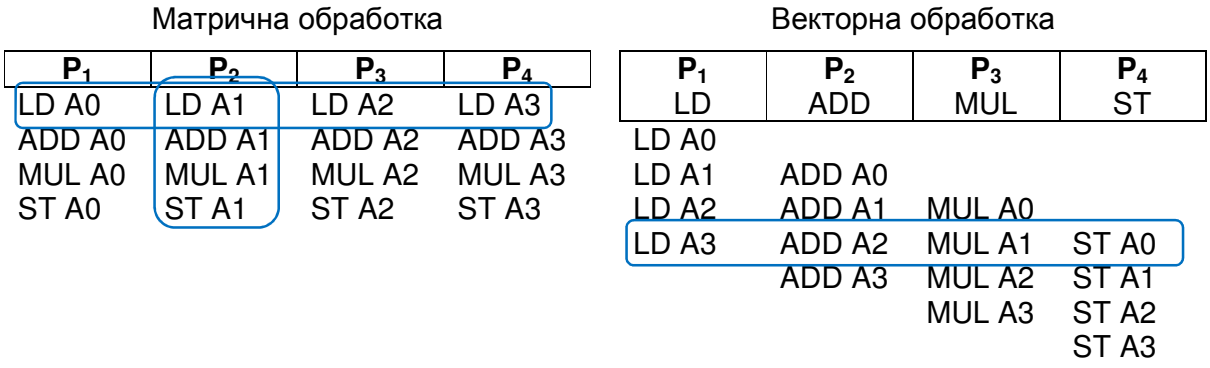


Фиг. 1.5 Единична инструкция оперира върху множество от данни

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

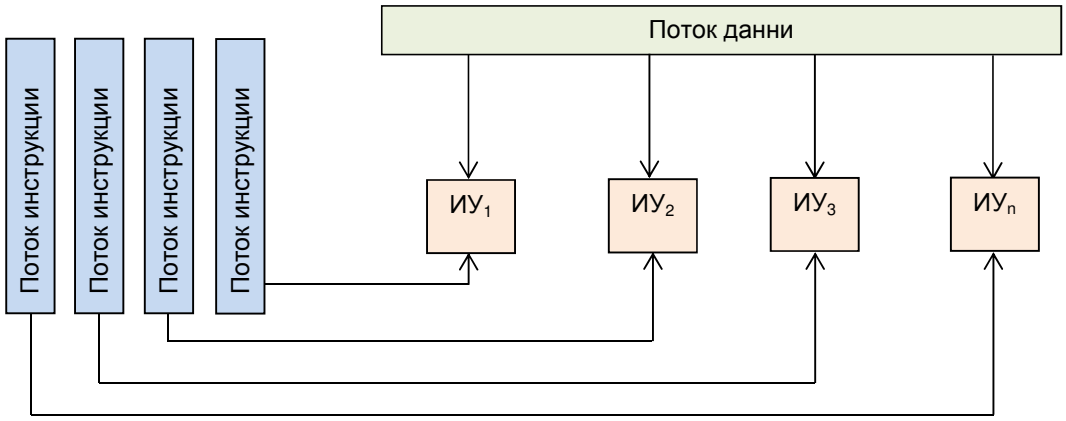


Фиг. 1.6 Матрично и векторно изпълнение

* 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

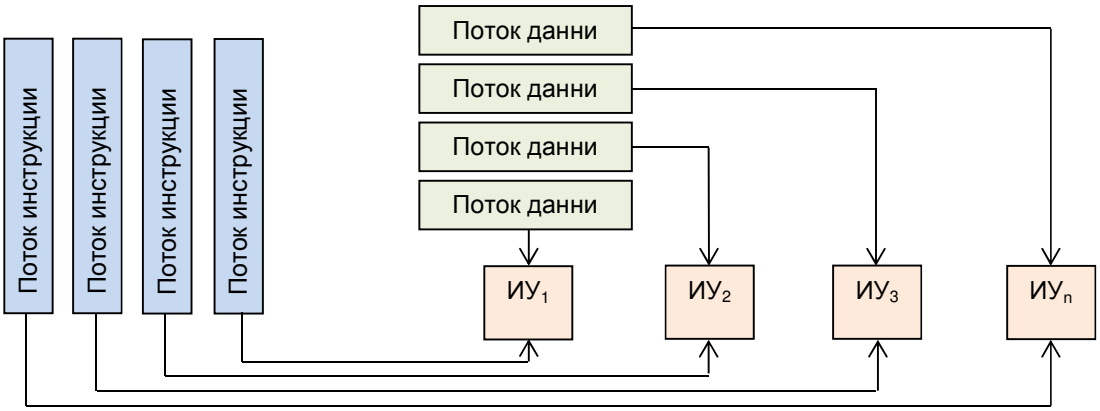


Фиг. 1.7 Различни инструкции оперират върху един елемент данни

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. 1.8 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, в случай на 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.

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.

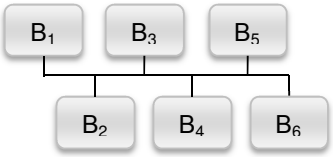
- Алгоритмите за комуникация – зависи как едно съобщение ще стигне от източника до приемника. Това може да стане по статичен или адаптивно променян маршрут.

- 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.

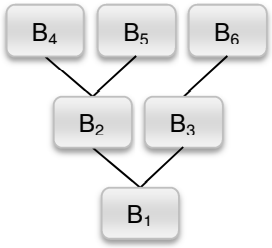
При топология от типа обща шина всички звена са свързани към една шина. Това прави модела най-лесен за изработка и евтин за осъществяване, когато имаме малък брой елементи, прикачени към шината (Фиг. 1.9). Недостатък на този модел е общият ресурс, ползван от всички елементи. При голям брой изчислителни 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.

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. 1.9 Разпределена шина

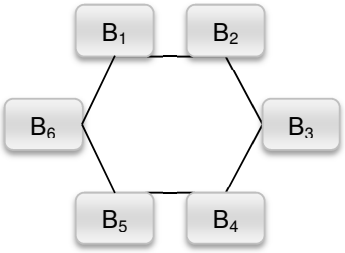


Фиг. 1.10 Обща шина

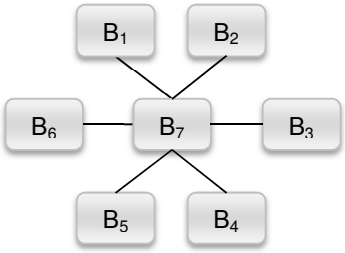


Фиг. 1.11 Дърво

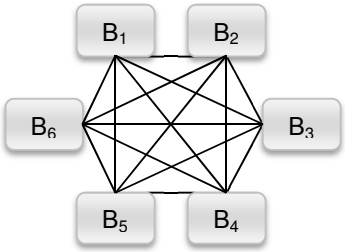
Предимствата на топология тип дърво са при специализиран тип задачи, изискващи локален обмен на данни – например на Фиг. 1.11 между възлите B4, B5 и B2. Те са евтини и лесни за изработка. При задачи, изискващи комуникация между всички възли, коренът на дървото (B1) се претоварва и забавя цялата система



Фиг. 1.12 Пръстен



Фиг. 1.13 Звезда

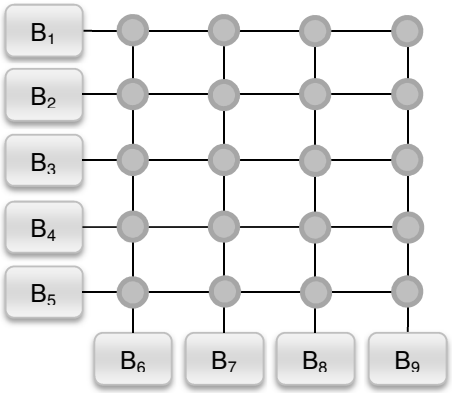


Фиг. 1.14 Напълно свързана

При топология от тип пръстен (Фиг. 1.12) 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. 1.14 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. 1.15.



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. Недостатък на матричната топология е цената за изработка, както и трудността при добавяне на голям брой възли.

Фиг. 1.15 Матрица

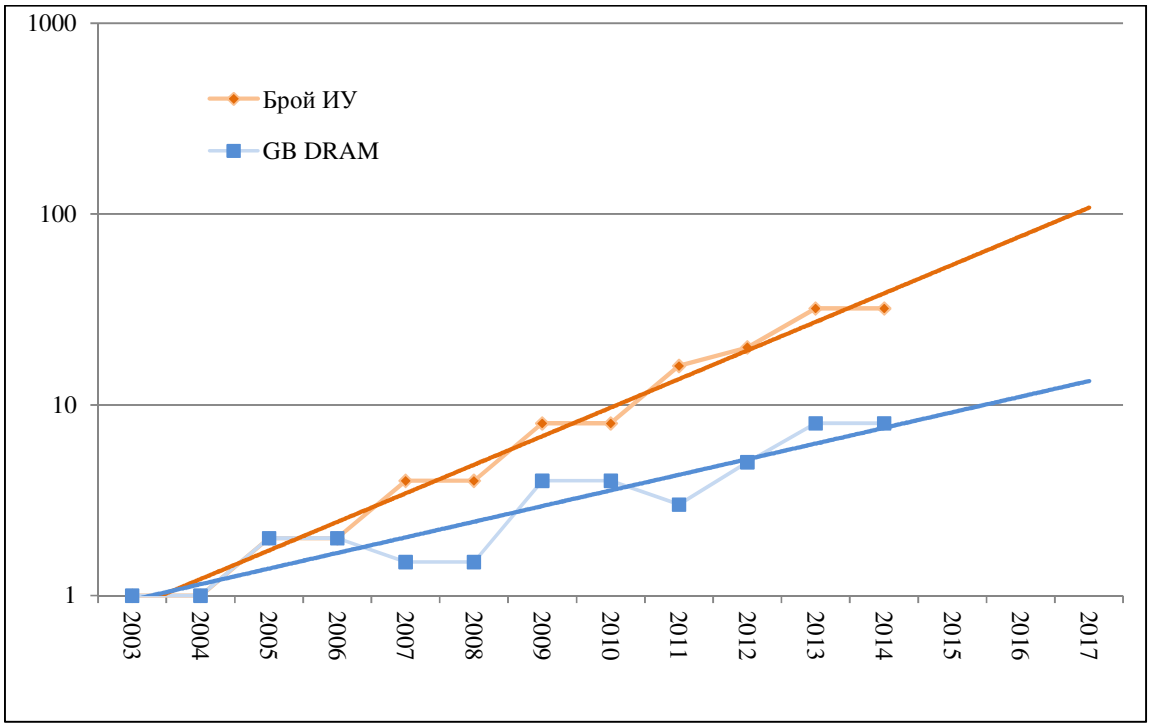
За подобряване на матричната топология са разработени модификации с поставянето на буфери между елементите или при комутационните звена, с които да се подобри пропускателната способност на мрежата. С поставянето на буфери сложността на мрежата се увеличава и оскъпява допълнително. Стремежът за изпълнение 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. 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.

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 [3] [5]. 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. 1.16 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.

ГЛАВА II
Архитектура базирана на изчислителни звена в паметта

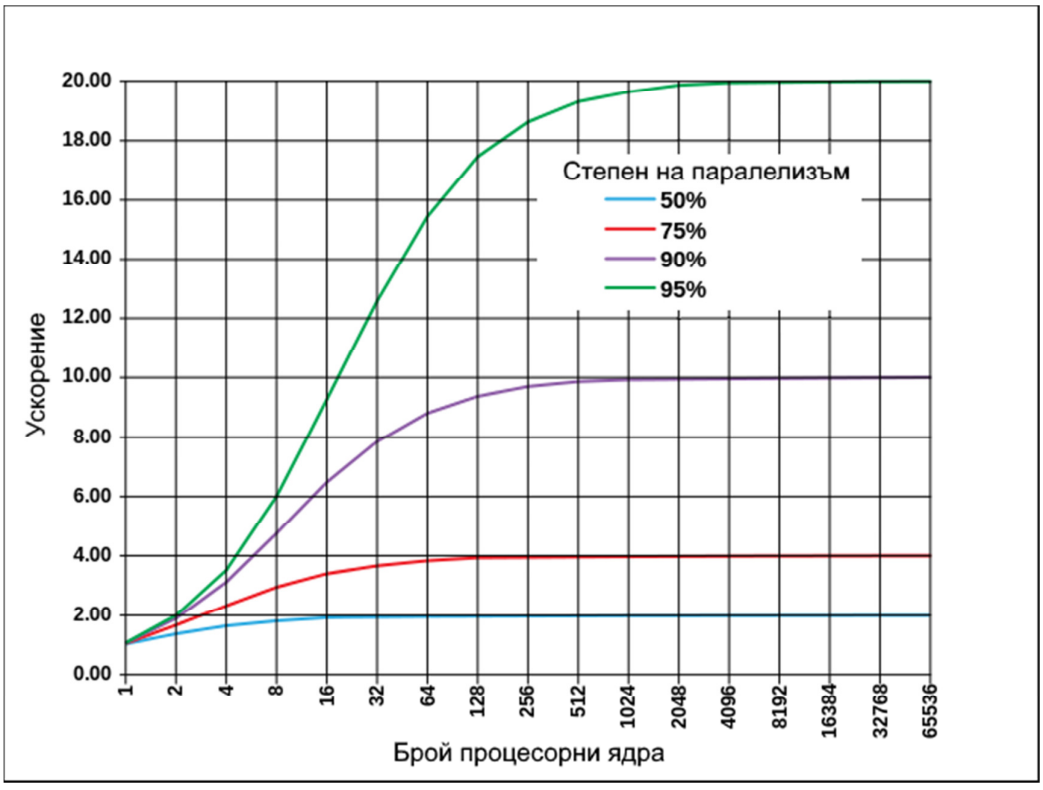
2.1 Текущи ограничения и недостатъци на многопроцесорните системи

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.

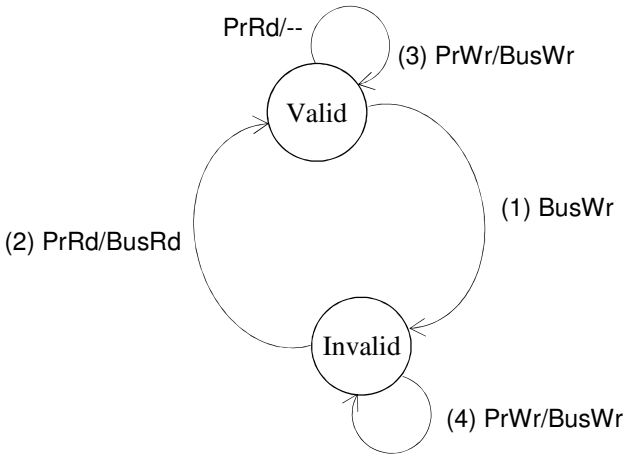
Комуникацията между различните сегменти води до допълнително забавяне. Не всеки път при сегментирането на едно приложение подобряваме времето за изпълнение. Както отбелязахме по-горе, сегментите може да имат нужда от общи променливи или ресурси, необходими на конкурентни сегменти. Така те не само трябва 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.

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.

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.

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; 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. 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:



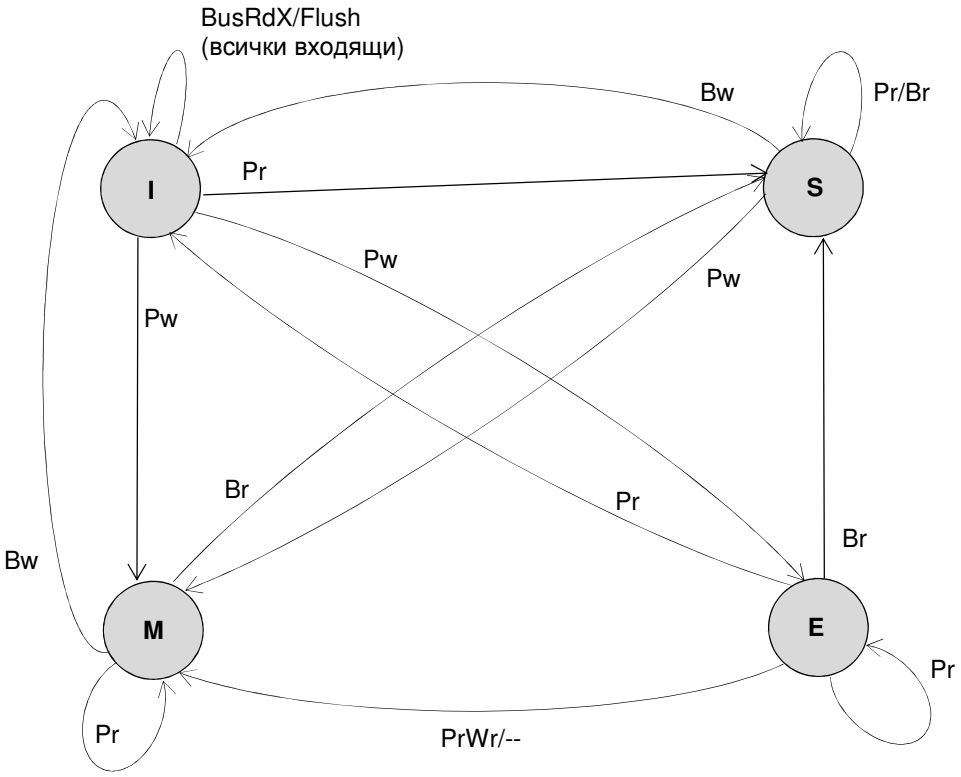
По общата шина разпространяваме:
PrRd – Processor Read – процесорът чете от локалната памет;
PrWr – Processor Write – процесорът записва в локалната памет;
BusRd – Bus Read – заявка за четене на локалната памет;
BusWr – Bus Write – заявка за запис на локалната памет.

Фиг. 2.2 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.

Подобрения на този модел са протоколите: MSI (Modified Shared Invalid, с добавено допълнително, трето състояние Modified), MESI (с допълнително състояние Exclusive) и MOESI (с допълнително състояние Owner). 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. 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.

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

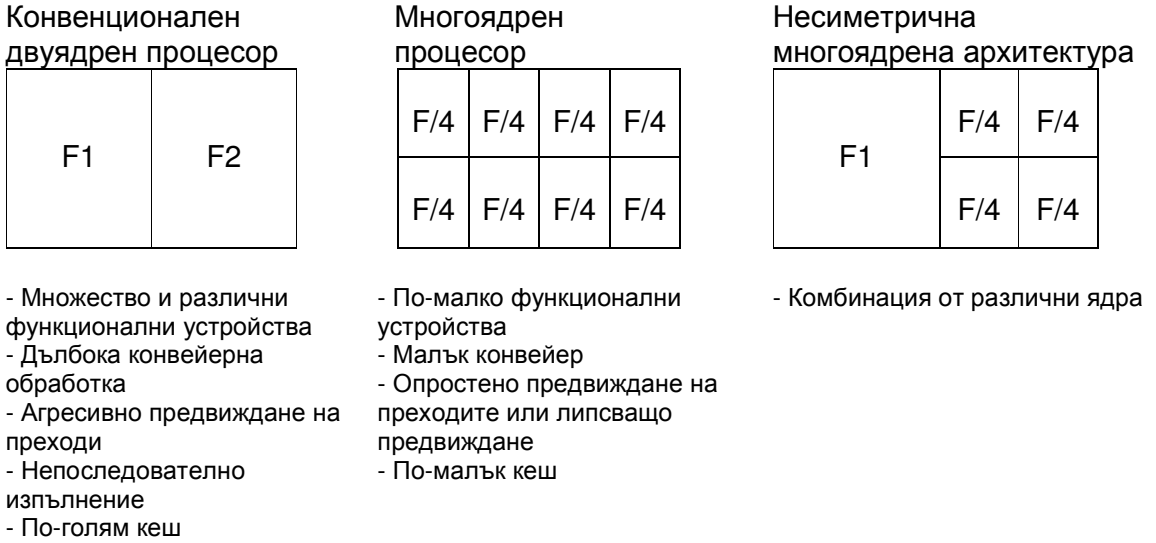
Преход от състояние E или M към състояние S се нарича „понижение“, защото преходът отнема правата на процесора да модифицира блока без съобщение по шината. Съответно промяна от S (споделено състояние) към състояние E или M е „повишение“, защото дава права на процесорното звено за „безмълвна“ промяна 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 се 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. 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 [6]. 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. При различни приложения сегментирането и обработването на отделните сегменти ще доведе до различни резултати, като понякога може дори да забави времето на изпълнение. Несиметричните многоядрени архитектури ще ни дадат възможност за различни решения на този проблем. Паралелните сегменти на приложението могат да бъдат изпълнени на много и по-малки ядра, докато изчакват изпълнението на последователната секция.



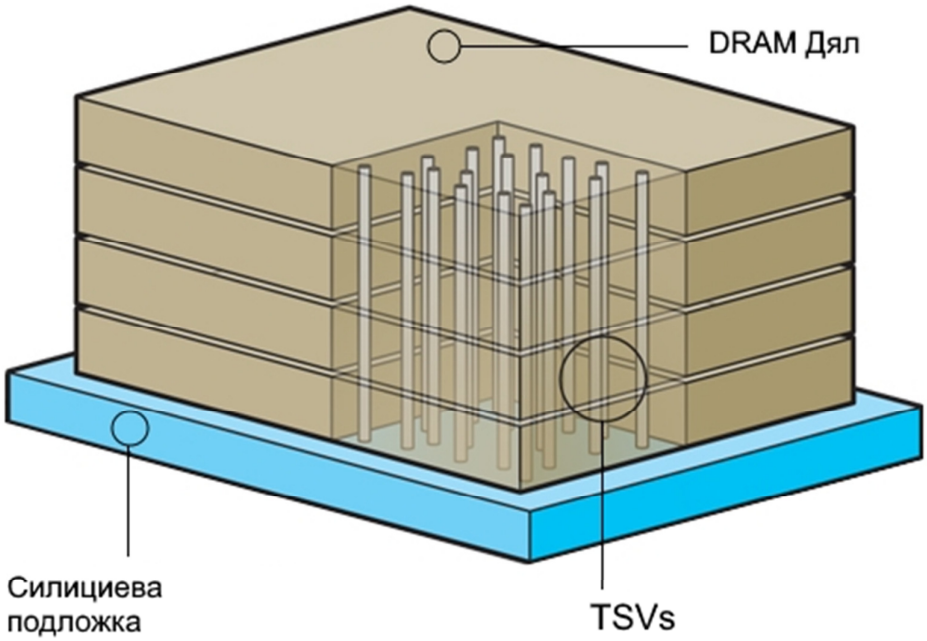
Фиг. 2.6 Несиметрични многоядрени архитектури

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 [7]. 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 слабата 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.

2.4 Изследване и предложения за използване на mPIM

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 хибридния куб вертикалните връзки са изградени вътре в пакета, което драстично съкращава електрическите пътеки. 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. Хибридната технология позволява разработването на изчислителни ядра, изградени върху първата силициева подложка.

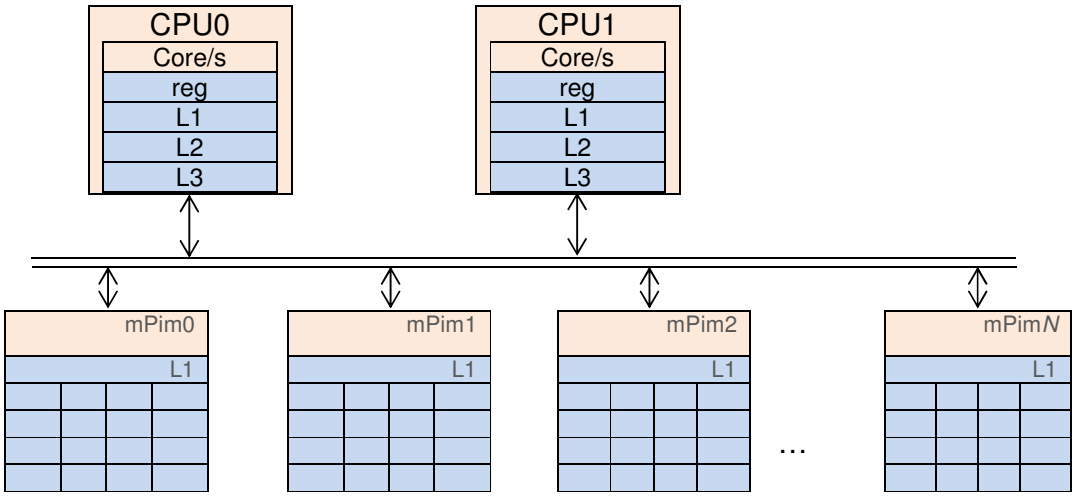


Фиг. 2.7 Свързване чрез TSV (through-silicon via) технология

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.

Съществуващата технологя предложена от HMC Tech, ще ни позволи разработката на множество малки и хетерогенни изчислителни елементи, директно прикрепени към паметта. При изследването на методите за ползване на mPIM ще се съсредоточим върху архитектура, съставена от основен конвенционален процесор или няколко процесора с помощните ресурси 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.

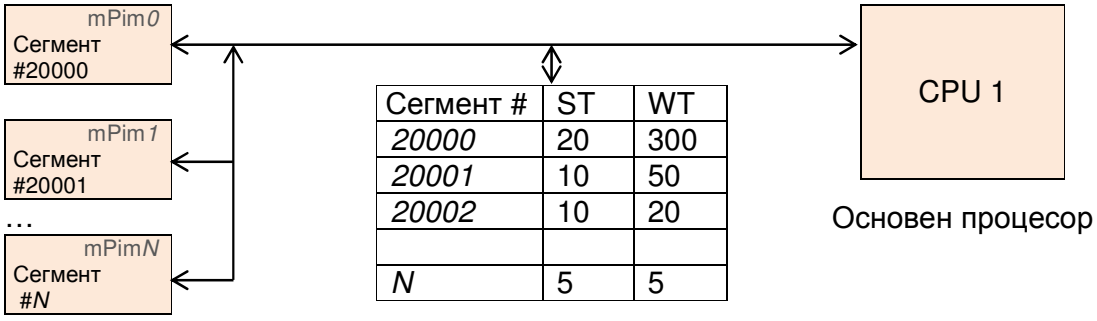


Фиг. 2.8 Многопроцесорна архитектура с изчислителни ресурси в паметта

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. В дисертационният труд са предложени и изследвани следните методи за използване на допълнителните ресурси:

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

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.

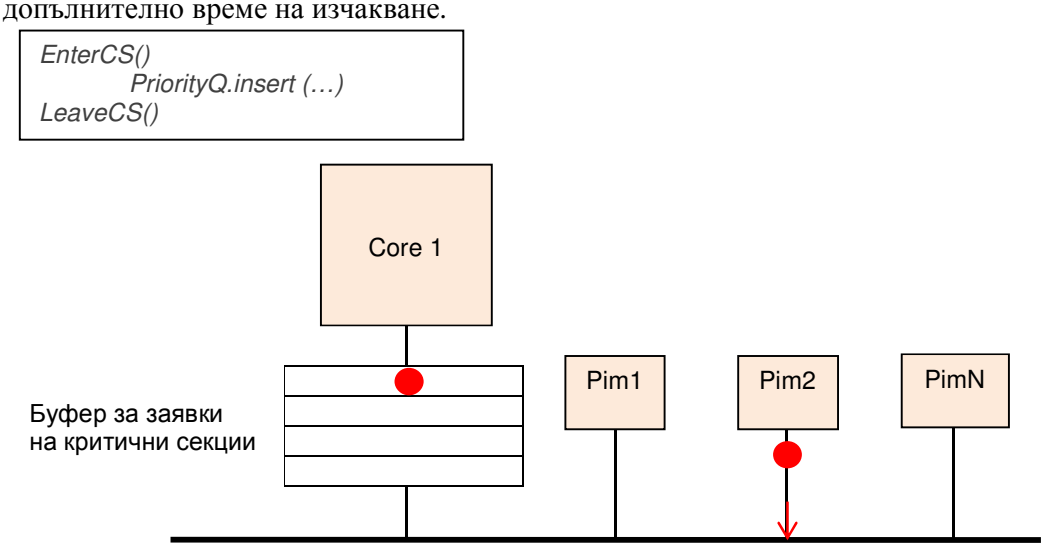


ST – Segment Time – Време за изпълнение на сегмента
WT – Wait Time – Комбинирано време на изчакващи сегменти

Фиг. 2.9 Динамичното идентифициране на непаралелизираната порция код

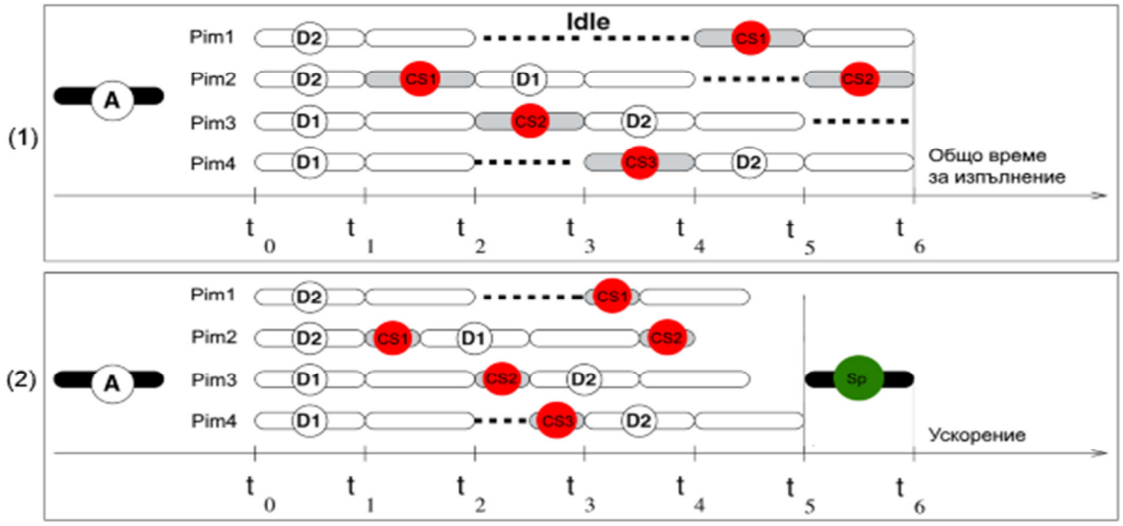
• Паралелна обработка на сегментите от mPIM и изпълнение на критичните секции от основния процесор

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. 2.10 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. 2.11 Разпределение на критичните секции към основния процесор

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.

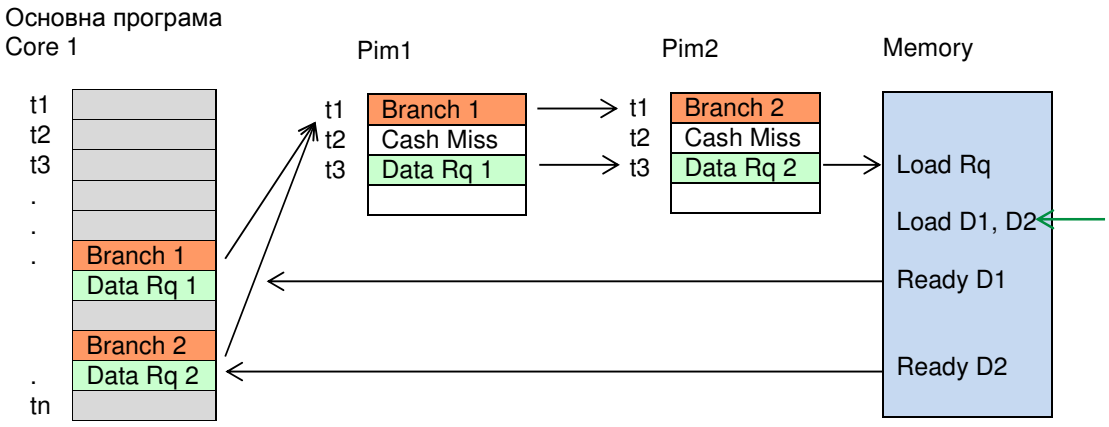
Ако в целия буфер на основния процесор имаме заредени няколко зависими критични секции, а едно от ядрата в паметта попадне на критична секция, независима от заредените, то ние можем да започнем нейното изпълнение паралелно в mPIM. Така ще постигнем паралелизация при изпълнението на независимите критични секции по същия начин на изпълнение, както при конвенционалните многоядрени архитектури. Като резултат кодът за изпълнение и съдържанието на една програма остава непроменен. Не са необходими специални модификации, тъй като съвременните операционни системи поддържат многоядрени архитектури и специализирано разпределение на различните ядра.

- 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. 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.

- 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.



Фиг. 2.12 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.

- Допълнителни и независими начини за използване на PIM: Използването на PIM ядро за компресиране на данните, изпращани по шината; Пакетиране на данните изпращани по шината на паметта.

2.5 Аналитичен модел на използването на изчислителни ресурси в паметта

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.

$$T = 1 - \%W_{PIM} \times \left\{ 1 - \frac{1}{N} \times \left[\frac{\tau_{pim} + LS \times (T_{Mpim} - \tau_{pim})}{1 + LS \times (T_{Cmp} - 1 + P \times T_{Mmp})} \right] \right\} \qquad (2.13)$$

$$\text{ако приемем, че } M \equiv \left[\frac{\tau_{pim} + LS \times (T_{Mpim} - \tau_{pim})}{1 + LS \times (T_{Cmp} - 1 + P_{MP} \times T_{Mmp})} \right] \qquad (2.14)$$

$$\text{то за T получаваме} \qquad T = 1 - \%W_{PIM} \times \left\{ 1 - \frac{M}{N} \right\} \qquad (2.15)$$

Където:

T = времето за пълен цикъл на PIM операция
%WPIM= процентно извършена работа от PIM
τ pim = такт на PIM
TMpim= PIM достъп до паметта
LS = средно време за четене и запис
TCmp = време за достъп до локален кеш на MP
P = стойност за брой пропуски в кеш паметта
N = броя на PIM възлите
и параметрите , 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:

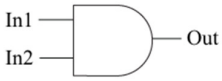
- Броя на PIM възлите;
- Частта от общата работа, която може да бъде подадена на PIM.

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

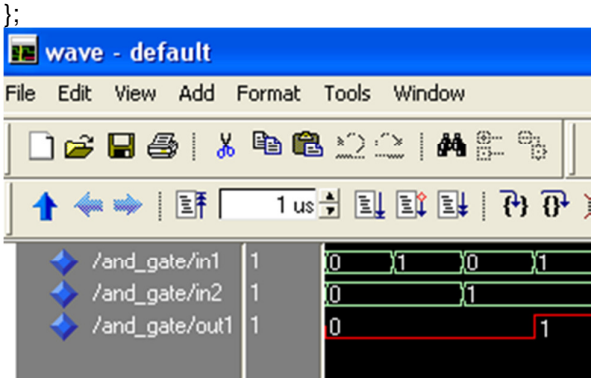


In1	In2	Out
0	0	0
0	1	0
1	0	0
1	1	1

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

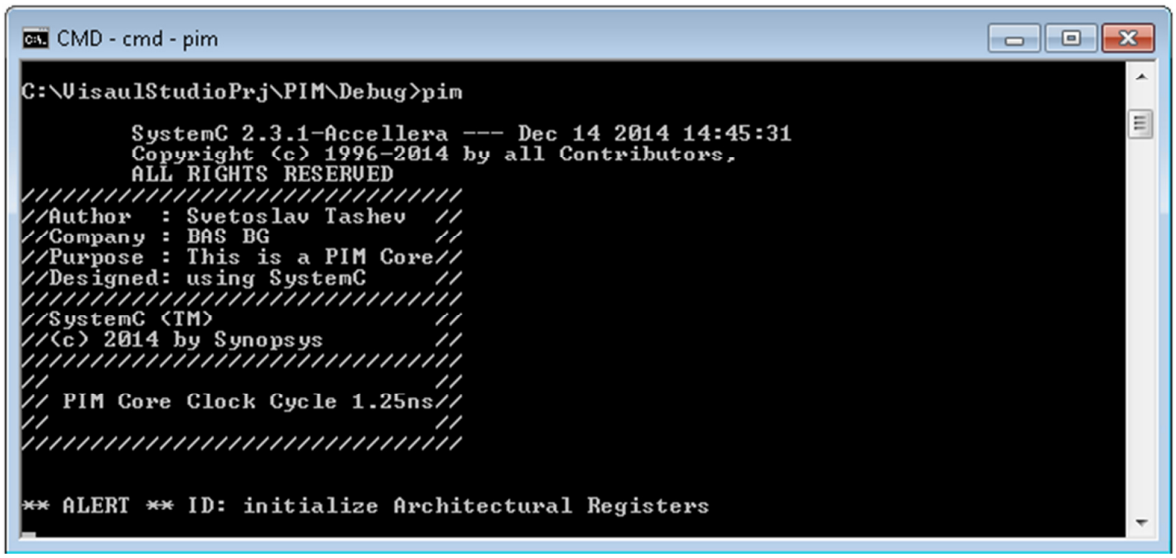
Тук е показан примерен дизайн на логически елемент "И" – "AND GATE" чрез ModelSim. Декларирането на този елемент чрез SystemC се извършва по следния начин

```
#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.

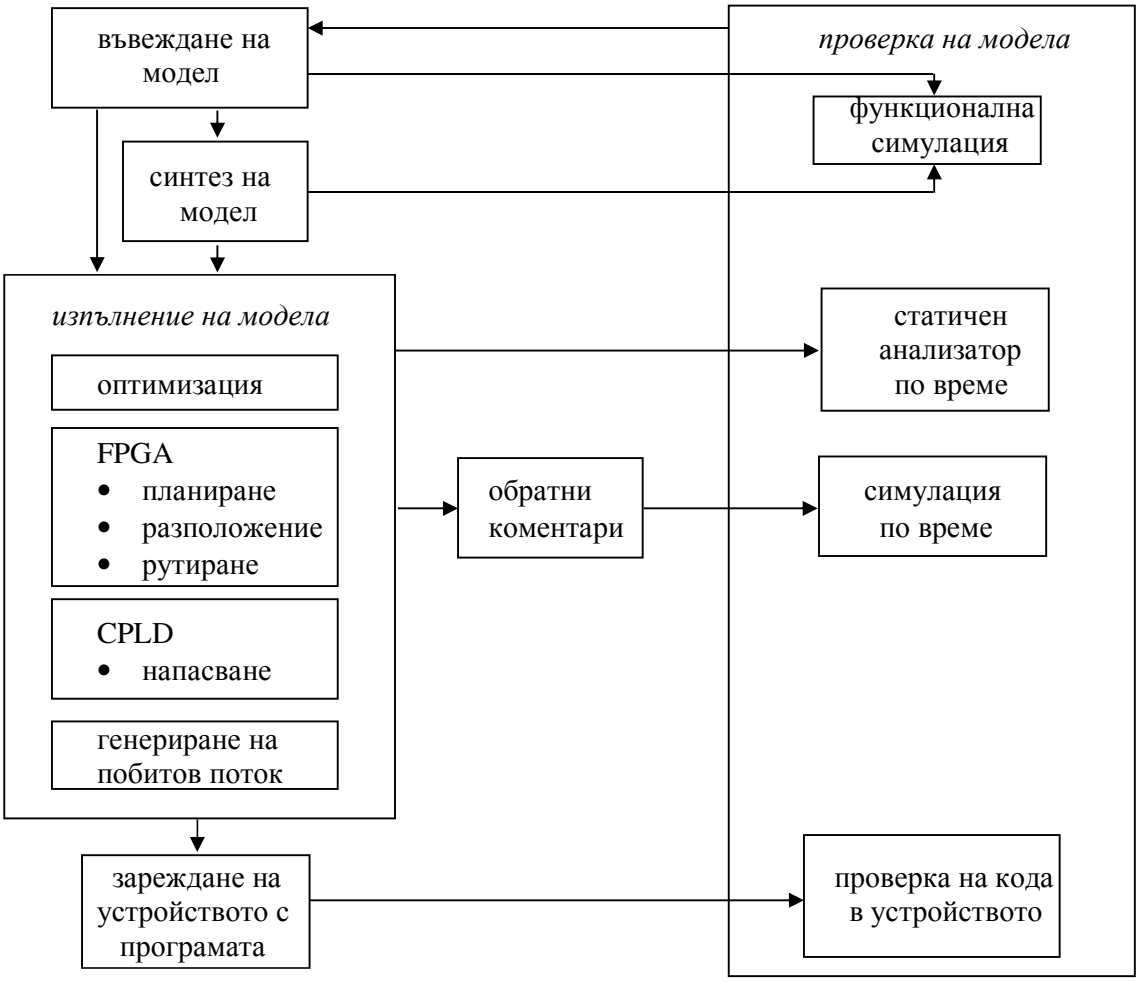


Фиг. 3.3 SystemC Model Debugging на PIM ядро

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

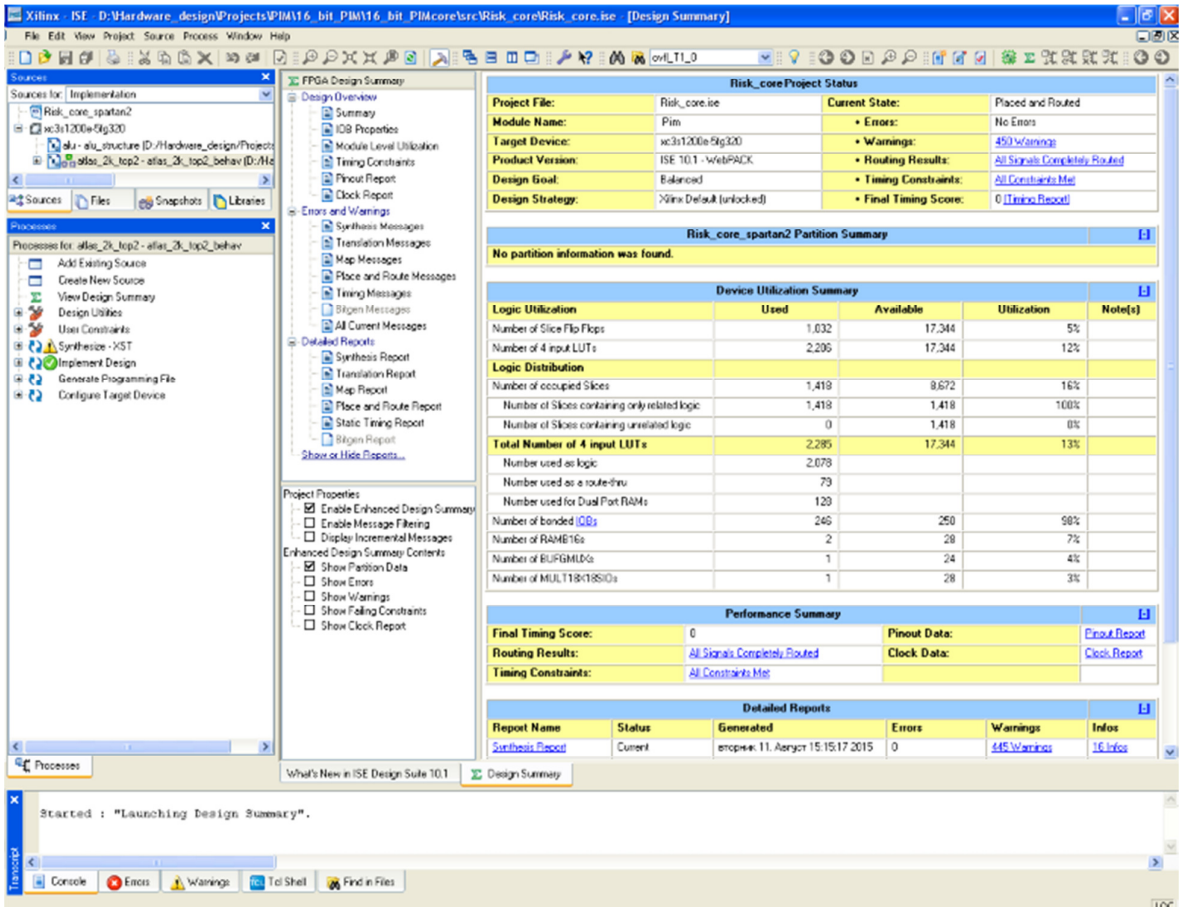
Изграждането на определен проект в средата ISE WEBPack е итеративен процес на симулация чрез проверка на логиката и проверка на поведението при състезание на сигналите, изпълняване и проверка на създаден модел до установяване на неговата коректност и финалното му завършване. Системата за разработка на 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

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

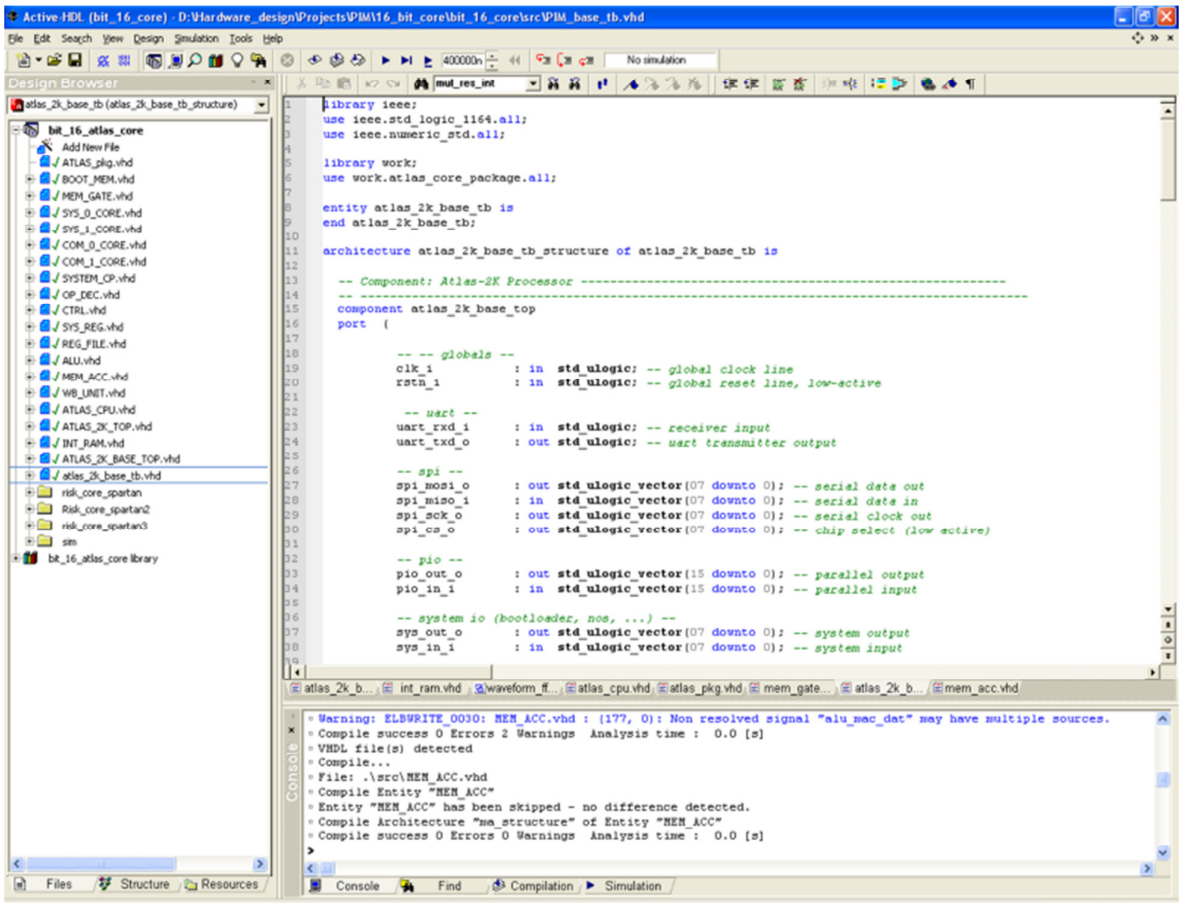
В своята същност ISE WEBPack представлява интегрирана среда за проектиране, изградена на модулен принцип, чиито зависимости се тълкуват от ISE и се показват като дървовидна структура. За едночипов дизайн това може да бъде един основен модул с други модули, включени към главния.

Design entry module - дава възможност за разработка на проекти, разработени 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 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;

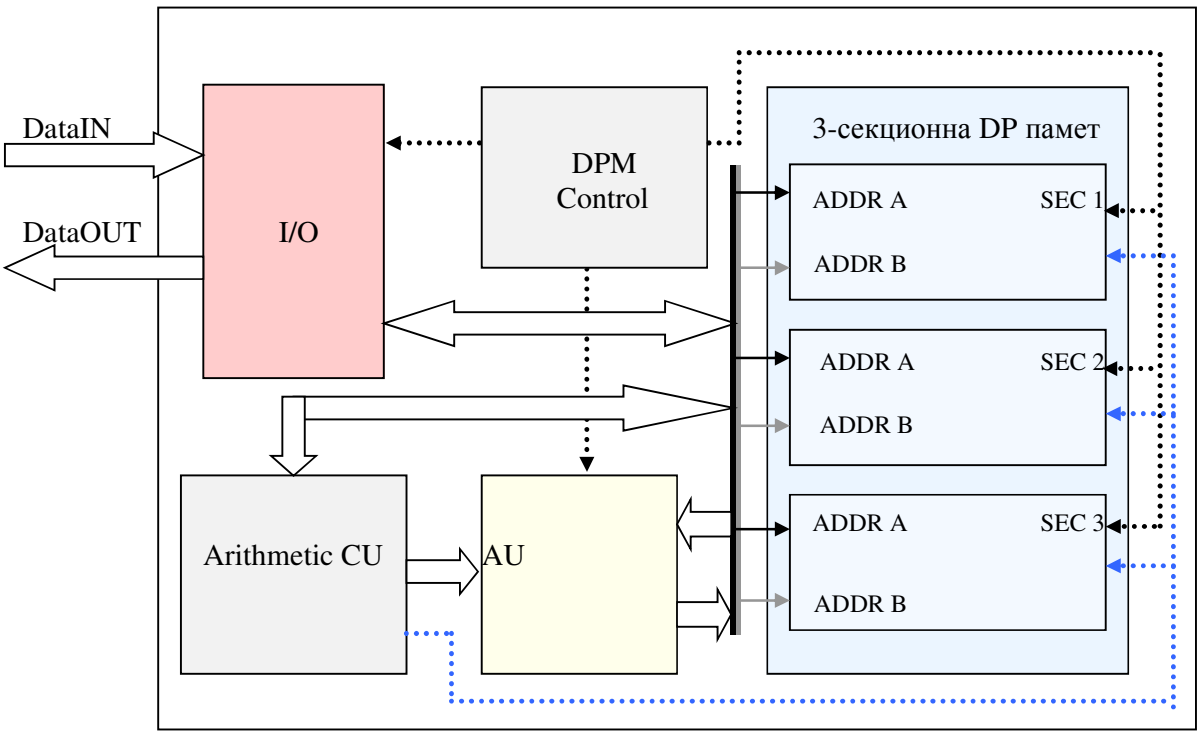
Arithmetic CU – управление на аритметичното устройство;

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.

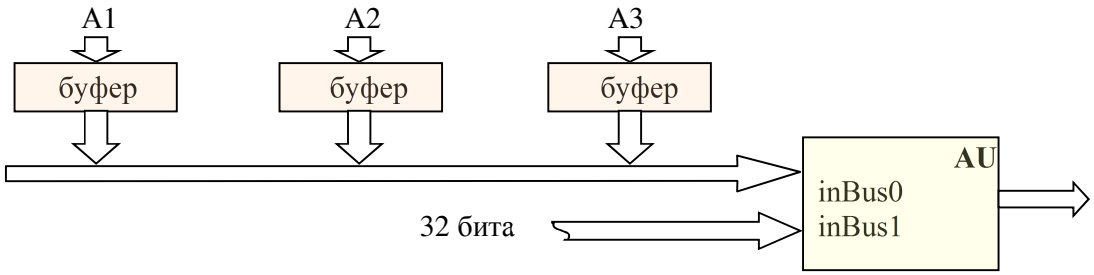
3.4 Реализация на различните PIM модули с HDL

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. 3.7 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. 3.8 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.

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

4.1 Избор на симулативно натоварване

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“ [9] 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:

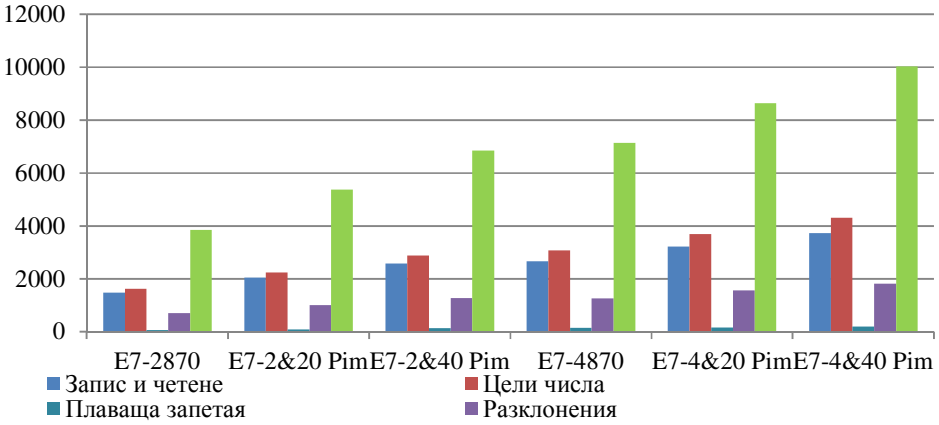
v31c2: E7-2870 симулативен модел на конвенционална система Intel Xeon E7-2870 с две гнезда и два процесора. Архитектура с 20 процесорни ядра.
v31c2P2: E7-2870 симулативен модел на конвенционална система Intel Xeon E7-2870 с две гнезда и два процесора, плюс 20 допълнителни PIM ядра.
v31c2P4: E7-2870 симулативен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

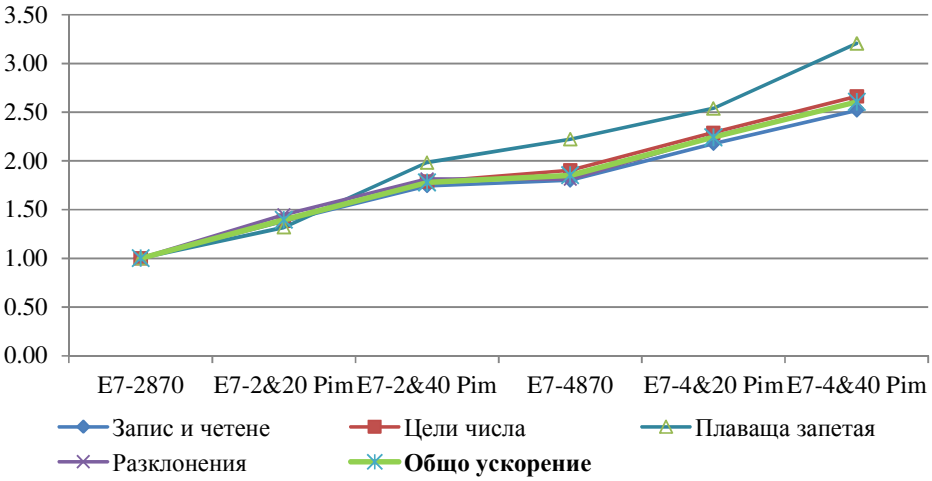
Общите комбинирани резултати за всички изчислени инструкции през времето на провеждане на серията тестове са следните:



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 [10].

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.

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.

Авторска справка

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. Предложена е промяна в основната диаграма на изпълнение на инструкциите.

3. Създаден е 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.

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 реализирането, 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.

Литература

1. http://www.intel.com
2. John P. Shen, Modern Processor Design: Fundamentals of Superscalar Processors, Electrical and Computer Engineering, McGraw-Hill Science, 2004
3. HP.com, Darrel D. Donaldson, AlphaServer 4100 Performance Characterization 2006
4. http://www.amd.com
5. Prof. Onur Mutlu, Carnegie Mellon, Computer Architecture 2015
6. Prof. Yale N. Patt, The University of Texas at Austin, Computer Systems
7. Prof. Yale Patt, Accelerating Critical Section Execution with Asymmetric Multi-Core
8. HMC Gen2 LOT Rev. 1.1 2014
9. Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ Processor 2011
10. TCP Benchmark Transaction Processing OLTP 2014
11. Intel, Metrics - Branch Mispredict node 529600 2015
12. Architectures, ASPLOS 2009
13. Dr Mike Murphy, Coastal Carolina University, Computer Science 2011
14. HP Labs, Disaggregated Memory for Expansion and Sharing 2009
15. Heterogeneous Chip Multiprocessors, 2005 vol.38 no.11
16. John L. Hennessy, Computer Architecture - A Quantitative Approach (4th edition), Morgan Kaufmann, 2006
17. Linda Null, The Esseцtials 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. nVIDIA 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. Richard Gerber, The Software Optimization Cookbook (2nd edition), Intel Press, 2006
24. http://www.xilinx.com
25. http://www.ieee.org/portal/site
26. http://www.apple.com
27. http://www.cray.com
28. http://en.wikipedia.org/wiki
29. http://www.ibm.com
30. http://lpsolve.sourceforge.net/5.5/
31. http://www.llnl.gov/
32. http://www.springerlink.com/home/main.mpx
33. http://www.verilog.com/
34. http://www.systemc.org/
35. http://lpsolve.sourceforge.ne
36. http://www.arm.com/products/CPUs/families/ARM9Family.html ARM9E Family of Embedded Processors
37. http://pages.cs.wisc.edu/wwd/rev4.pdf
38. http://www.broadcom.com/collateral/pb/2702-PB02-R.pdf BCM2702: High Performance Mobile Multimedida Processor