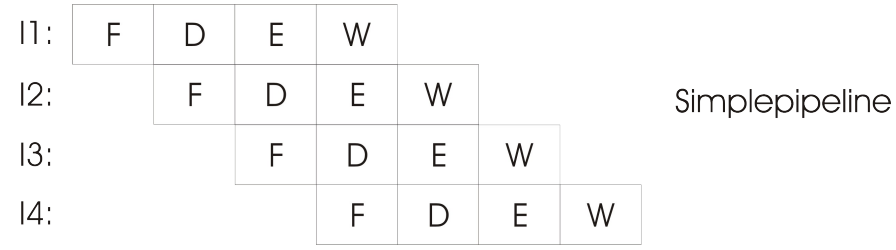
Сравнение на конвейерен, суперконвейрен и суперскаларен подход

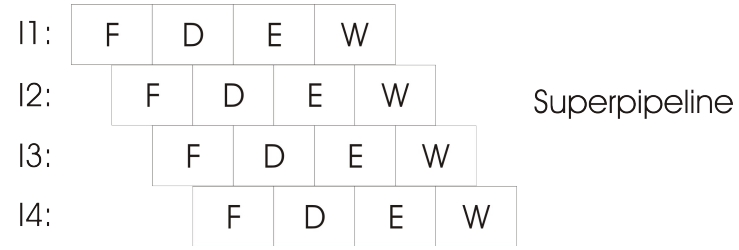
Конвейерен подход – изпълнението на инструкциите тук е последователно и само по една на такт. Метода датира началото си още средата на миналият век. Ако вземем предвид нормален конвейер от четири такта – извличане (Fetch), декодиране (Decode), изпълнение (Execute) и запис (Write back), то метода на изпълнение на инструкциите ще бъде следният:



фиг.3.16. Конвейрно изпълнение на инструкции

Фигура 3.16 ясно показва неефективността на конвейрното изпълнение – за цели четири такта, ще е изпълнена само една инструкция и следващата я ще е стигнала до фаза изпълнение. Тази техниката е довела до търсене на по-бързи и по-ефективни методи за изпълнение на инструкциите.

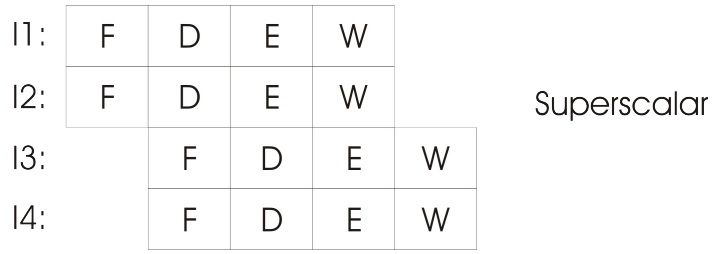
Следващият подход е суперконвейрният. Фазите, през които минава всяка инструкция са отново четири, но тук вече започва да се говори за паралелно изпълнение на подаваните:



фиг.3.17. Суперконвейрно изпълнение на инструкциите

При този подход за четири такта като резултат изпълняват се една и половина инструкции. Паралелизма е налице и дълго време този тип архитектура се ползва от скаларните процесори.

Коренна промяна имаме при изпълнението на суперскаларните процесори.



фиг.3.18. Суперскаларно изпълнение

Ясно вече си вижда паралелизма на ниво инструкции – метода е комбинация от предходните два – за четири такта, вече има две изпълнени инструкции, като следващите двойки чакат своя ред след тях. Метода довежда до бурно развитие на суперскаларните процесори и особено търсенето на други техники за повишаване бързодействието и производителността на процесора.

Паралелизъм на ниво инструкции. Зависимости между инструкциите

Във всяка програма, инструкциите обикновено зависят една от друга по такъв начин, че конкретна инструкция не може да бъде изпълнена, докато предходна или дори две, или три предходни инструкции не са изпълнени. Прост пример можем да дадем, че зависимост съществува ако инструкция се нуждае от резултати на предишна като източник операнд.

Съществува три възможни варианта на зависимост между инструкции. Ако последователни инструкции са зависими една от друга по данни имаме *зависимост по данни*(data dependencies). *Зависимост по управление*(control dependencies) са следващият вид, свързани с условен или безусловен branch. И последни, но не и по значение, е *зависимост по ресурси*(resources dependencies), когато на две инструкции са необходими едни и същи ресурси за изпълнението им.

- **Data dependencies – концепция на зависимост по данни**

Нека са дадени две инструкции i_k и i_l от една и съща програма, където i_k предхожда i_l. Ако i_k и i_l имат общ операнд, бил той от памет или регистър, те са зависими по данни една от друга, освен когато общата операнда се използва и в двете инструкции като source operand. Очевиден пример за такава зависимост е когато i_l използва резултати от i_k като source operand. При последователно изпълнение зависимост между данните не съществува, тъй като инструкциите се изпълняват стриктно в определен ред. Но същото не важи, когато инструкциите се изпълняват паралелно, както при паралелни архитектури на

ниво инструкции(ILP). Тогава вече откриването и разрешаването на тези зависимости става основна задача.

В зависимост от типа на изпълнението имаме зависимост при последователни инструкции (in straight-line code) и зависимости между инструкции в цикъл (in loops). Първият тип зависимости се разделят на три вида – RAW(Read after Write – четене след запис), WAR(Write after Read – запис след четене) и WAW(Write after Write – запис след запис), докато инструкциите обхванати в цикли – рекурентна(recurrences).

Data dependencies in straight-line code

Примерите са дадени с регистрови данни, като данните от паметта могат да се интерпретират по абсолютно същият начин.

RAW dependencies – Четене след запис.

Нека вземем две асемблерни инструкции:

i1: load r1, a;

i2: add r2, r1, r1;

В примера инструкция i2 използва r1, като source. Като резултат, i2 не може да бъде коректно изпълнена докато r1 не е зареден от инструкция i1. Следователно, i2 е RAW зависима от i1.

Най-общо казано, инструкция е RAW зависима от друга, когато имаме четене след запис от един и същ адрес. Това означава, че инструкцията, която чете се нуждае от резултати на предишната и затова не може да се преодолее и инструкциите трябва да се изпълнят в определения ред. RAW зависимостите освен това могат да бъдат – load-use и define-use dependency.

Load-use зависимости възникват, когато данните в източника операнд не са заредени и първо трябва да се зареждат, за да се изпълни инструкцията. Пример:

i1: load r1, a;

i2: add r2, r1, r1;

Другата RAW зависимост, define-use, възниква когато източника операнд е дефиниран непосредствено в предходната инструкция:

i1: mul r1, r4, r5;

i2: add r2, r1, r1;

WAR зависимост – Запис след четене.

Нека разгледаме следните инструкции:

i1: mul r1, r2, r3:

i2: add r2, r4, r5;

В този случай i2 прави запис в r2, а пък i1 използва r2 като източник. Ако i2 се изпълни преди i1 то първоначалното съдържание на r2 ще бъде презаписано с информация, която не е необходима на i1, откъдето идва грешен резултат при изпълнението й. WAR зависимостта съществува между две инструкции, ако втората записва резултата си в някои от източниците на първата инструкция. WAR и WAW зависимостите са фалшиви и могат да бъдат елиминирани чрез **register renaming**, като сменим резултатния адрес на запис на втората инструкция с друг адрес, който не се използва. И тогава проблемът в горния пример може лесно да се реши, като просто преименуваме r2 от инструкция i2 в r6, например:

i1: mul r1, r2, r3;

i2: add r6, r4, r5;

Фалшивите зависимости могат сериозно да намалят производителността на ILP-процесорите, затова се използват различни техники за откриването им и тяхното преодоляване.

WAW зависимост – Запис след запис

Две инструкции са WAW зависими, ако и двете използват един и същ регистър за запис на резултатите си, като в следния пример:

i1: mul r1, r2, r3;

i2: add r1, r4, r5;

Този вид зависимост, очевидно, е отново фалшива и може да бъде преодоляна - по същия начин като WAR зависимостта – чрез преименуване на регистъра.

- **Зависимост по управление**

Нека разгледаме следната последователност от инструкции:

mul r1, r2, r3;

jz zproc;

sub r4, r1, r1;

:

zproc: load r1, x;

В този пример, изпълнението зависи от резултата на умножението. Това означава, че инструкциите които следват след условния преход са зависими от

него. Условните преходи мога сериозно да намалят производителността при паралелно изпълнение на инструкциите. Затова е изключително важно да се изследват колко често се появяват.

Статистически проучвания, показват че програмите с общо приложение се държат по доста различен начин от научни програми, разглежданите вероятности за преходи – били те условни или не. Под програми с общо приложение се вземат предвид компилатори, операционни системи и програми без сложни изчислителни задачи. Програми с общо приложение имат доста висок процент на разклонения – стигат до 20-30 %. В контраст на тях са научни/техническите програми, които поддържат доста нисък процент – 5-10%. Отношението на условните преходи изглежда доста стабилно в различните програми и остава в диапазона 75-85 %. В следствие на това, очакваната честота на условни преходи във програмите с общо приложение е около 20 %, докато в научните програми е само 5-10 %. Като извод можем да кажем, че при ILP-процесори, зависимостите по инструкции представляват доста сериозна пречка в ускоряването на бързодействието при програми с общо приложение, отколкото при научните.

- **Зависимост по ресурси**

Инструкция е зависима по ресурси от предходна инструкция, ако се нуждае от хардуерен ресурс, който е зает от друга инструкция. Като пример може да се разледа следната последователност от инструкции:

i1: div r1, r2, r3;

i2: div r4, r2, r5;

Ако имаме само една логика за деление, то не могат да се изпълнят инструкциите паралелно, а втората ще трябва да изчака завършването на първата, за да се възползва от ресурса.

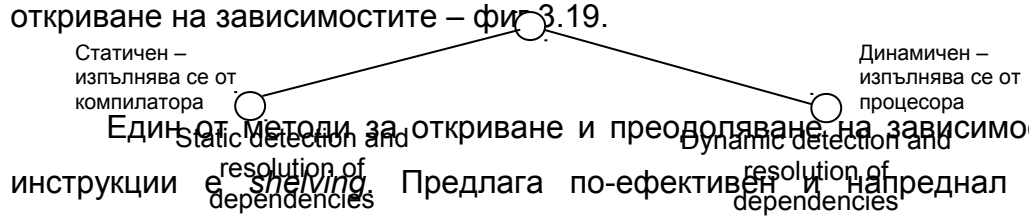
Техники преодоляващи зависимости между инструкции

1. Instruction Scheduling – Подредба на потока от инструкции

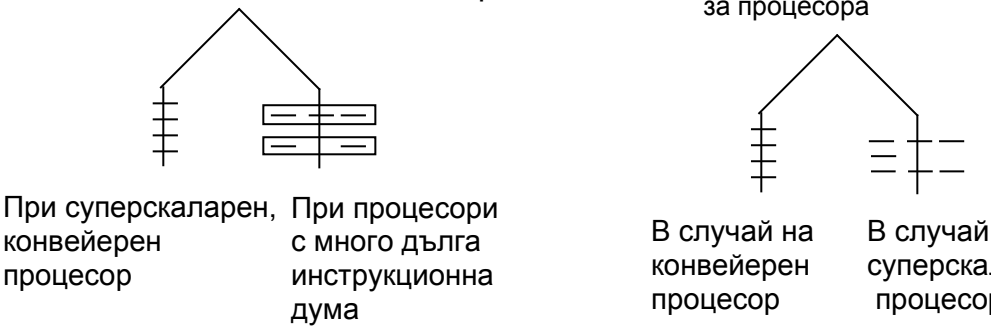
Когато инструкциите се изпълняват паралелно, често е необходимо зависимостите между инструкциите да се разкрият и разрешат. Въпросът е, дали тези задачи да се изпълняват от компилатора или от процесора. Двата основни подхода са наречени – статичен и динамичен. Статичното откриване и преодоляване се реализира от компилатора, който избягва зависимостите с

метода на преподреждане на кода. Идеята е след групирането да се получат инструкции, които са независими помежду си и следва да бъдат изпълнени паралелно.

За разлика от първия подход, вторият (динамичния) се осъществява чрез процесора. Той поддържа два прозореца, през които минават инструкциите. Прозореца за подаване на инструкции (issue window) съдържа всички предварително извлечени (prefetched) инструкции, които се очакват да се изпълнят в следващия такт, докато инструкциите които са още в изпълнение и на които резултати още не са получени се помнят в изпълнителният прозорец (execution window). По време на всеки цикъл, всички инструкции в прозореца за подаване на инструкции се проверяват за зависимости по данни, управление и ресурси, като се вземат предвид разбира се и инструкциите в изпълнение. Като резултат от проверките се стига до една или повече независими инструкции, които ще се насочат за изпълнение към изпълняващите конвейри(EUs). Трябва разбира се да подчертая, че това е само един от възможните методи за откриване на зависимостите – фиг 3.19.



Един от методи за откриване и преодоляване на зависимостите между инструкции е *shelving*. Предлага по-ефективен и напреднал подход при подаването на независими инструкции. Shelving-а използва специално създадени за целта инструкционни буфери, наричани още shelving буфери, които са разположени пред всеки конвейр. При този подход се разделя проверката за зависимости от блокирането на инструкциите и ги подава в shelving буферите, докато операндите им бъдат разрешени и чак тогава ги препраща към конвейрите за изпълнение. Когато някакъв конвейр се освободи, съответният му shelving буфер веднага проверява за готови за изпълнение инструкции. Някоя от тях се избира според политиката на извличане и бива пренасочена към изпълняващите конвейри.



При суперскаларен, конвейерен процесор

При процесори с много дълга инструкционна дума

В случай на конвейерен процесор

В случай на суперскаларен процесор

Код на
програмата

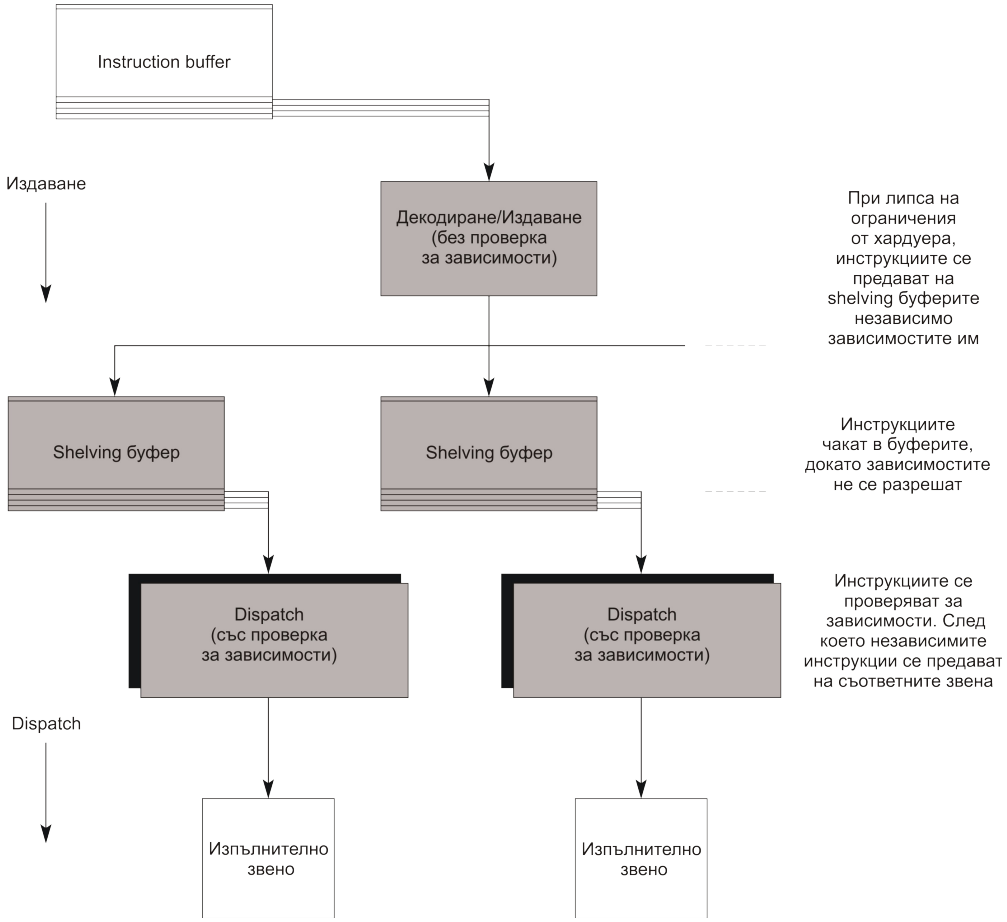
фиг.3.19. Основни принципи на подредба на потока от инструкции

Shelving

Методът се използва в съвременните процесори заедно с техниките за преименуване на регистри и спекулативно изпълнение на преходи, при което единствените зависимости, които остават за проверка преди подаването на инструкциите са тези по данни (RAW dependencies). Така инструкциите, които престояват в shelving буферите стават готови за обработка, когато всички необходими операнди са в наличност.

Shelving-а може да бъде, според целите на съответната архитектура, частичен или пълен(фиг.3.20). Като под частичен се разбира, че само конкретен вид инструкции ще преминават през shelving буфери, а останалите не; докато при пълния – всички инструкции следват идеята на shelving-а.

Видът на буфери също играе роля в концепцията на shelving-а. Разграничават се самостоятелни буфери, които се делят на индивидуални, групови и централизиран, според това по колко конвейра обслужват, съответно – един, няколко или всички налични; доста по различен подход са комбинираните буфери, които се използват за преименуване и пренареждане – тук ReOrder Buffer-а (ROB) се грижи за последователността на инструкционния поток и осигурява shelving и регистъра за преименуване. Въпреки комплектността на последният вид определено метода е доста ефективен и широко използват и днес.



фиг.3.20. Схема на работа на shelving буферите

Последователността на потока от инструкции се запазва благодарение на метод “преподреждане на инструкциите”.

Register Renaming – регистъра за преименуване

Основна техника, използвана за преодоляване на фалшивите зависимости по данни – WAR и WAW. Методът предполага формат за всяка инструкция от три операнди. Нека разгледаме следния пример:

add r1, r2;

интерпретацията на израза е

r1 ← (r1) + (r2)

При дву-операнди формат резултата се записва обратно на мястото на някоя от източниците операнди, както е в предходния пример. Именно за преименуването се използва регистър различен от използваните източници на операнди, в нашия случай, да кажем r22. След преименуването имаме:

r22 ← (r1) + (r2)

Вече няма опасност от фалшива зависимост, защото така няма опасност да изгубим стойностите на един от двата операнда.

Преименуването на регистъра може да се осъществи както статично, така и динамично. При статичното – процедурата се изпълнява по време на компилирането, и обратно при динамичното – по време на изпълнението. В началото, методът се е реализирал частично в процесорите, само за определен тип инструкции, но по късно производителите бързо преминават към пълна му реализация. Определено типа на буферите за преименуване оказват най-голям ефект от работата на метода. Ще разгледам примери, използващи само един регистров файл за всички видове инструкции.

Техники за ускоряване производителността на съвременните процесори

За да се постигне висока производителност съвременните компютърни архитектури използват две форми на паралелизъм – паралелизъм на ниво инструкции (ILP) и паралелизъм на ниво нишки (TLP). Въпреки че, те имат различна гранулярност те са фундаментално идентични, защото откриват независими инструкции, които могат да бъдат изпълнени паралелно и по този начин използват паралелния HW. Така наречените wide issue суперскаларните процесори експлоатират ILP чрез изпълнение на множество инструкции от еднонишкова програма за всеки такт. Мултипроцесорите откриват TLP, чрез паралелно изпълнение на различни нишки върху няколко процесора. Но нито един от двата подхода не е способен да се адаптира към динамичното

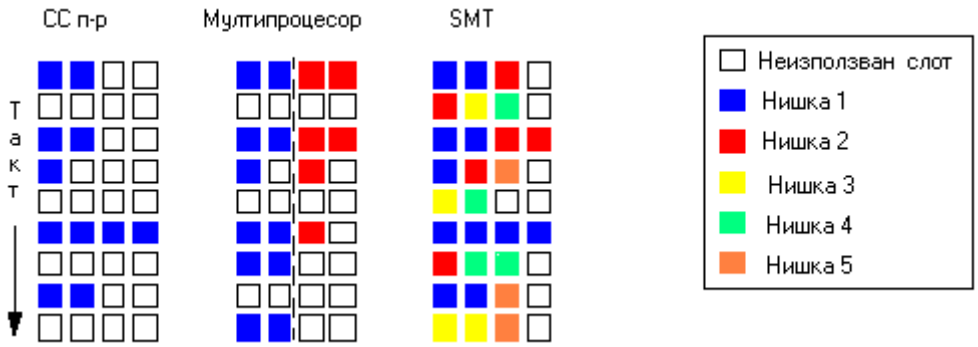
изменение на нивата на ILP и TLP, защото HW комбинация е насочена твърдо към един от двата типа паралелизъм. Недостатък при мултипроцесорите е, че ако имаме недостатъчно TLP в програмата, някои от процесорите ще останат неизползваеми. От друга страна, суперскаларният процесор изпълнява една нишка и ако имаме недостатъчно ILP, голяма част от ресурсите ще остане неизползвани – особено тези, които са свързани с HW за едновременно подаване.

Simultaneous Multi-Threading (SMT) или т.нар. HyperThreading заимства предимствата на двата подхода, като позволява много нишки да се съревновават за разпределението на дадените процесорни ресурси всеки цикъл. Едно от ключовите предимства, когато изпълняваме паралелни приложения е способността на SMT процесора да използва двата паралелизма едновременно, като позволява няколко нишки да споделят процесорните ресурси едновременно (*simultaneous*). Когато програмата има само една нишка, т.е. липсва TLP, всичките ресурси на процесора ще бъдат предоставени на тази нишка, а когато съществува по-голям TLP (има няколко нишки) – той ще компенсира липсата на ILP във всяка нишка. SMT процесорът може уникално да експлоатира които и да е тип паралелизъм, по този начин използва функционалните устройства много по-ефективно и постига по-голяма пропускателна способност, следователно значително ускоряване на програмата.

Изследванията в дисертацията са насочени към паралелно изпълнение при SMT архитектура. Предишни изследвания са правени като е използвано мултипрограмно натоварване (*workload*) с цел оценка потенциала на SMT архитектури. Едно от основните предимства на SMT е това, че той може да бъде получен от SS процесор с малки изменения в архитектурата и както показват изследванията това води до повишаване на производителността спрямо *wide issue* суперскаларен процесор. Мултипрограмното натоварване доставя много TLP, защото всяка нишка принадлежи на изцяло различно приложение.

Както вече споменах обект на мои изследвания са паралелните програми, т.е. една програма е разделена на нишки, които се синхронизират и споделят данните. Тези програми имат по-различни изисквания към SMT, отколкото мултипрограмното натоварване. В частност сравнявам SMT с CMP –

това са small scale on chip multiprocessors и оценявам способността да откриват ILP и TLP(фиг.3.21).



фиг.3.21. Различни подходи

При суперскаларните процесори през по-голяма част от времето, повечето ресурси на системата не се използват, тъй като програмата има ниско ниво ILP. При мултипроцесорите (CMP) могат едновременно да се изпълнят няколко нишки върху различни процесори, но производителността им е ограничена от фиксираното разделяне между процесорите. От фиг.3.21 се вижда, че има недостатъчен TLP в програмата, т.е. някои процесори ще останат неизползвани. Друг недостатък при мултипроцесорите е използването на процесори с ниско ниво на ILP. SMT процесорът преодолява недостатъците на предишните два подхода, като има способността да открива едновременно ILP и TLP. В случаят, при SMT може да се подават четири инструкции от четири различни нишки едновременно. Динамично се разпределят ресурсите между нишките и когато има само една нишка всички ресурси се предоставят на нея, в противен случай високото ниво на TLP ще се компенсира липсата на ILP във всяка една нишка и по този начин няма да имаме празни слотове във базовия суперскаларен процесор.