Конвейерно изпълнение в процесор със система инструкции MIPS

Съдържание:

1. Преглед на MIPS;
2. Изпълнение на инструкциите;
3. Конвейерно изпълнение;
4. Състезания между инструкциите в конвейера.

1. Преглед на MIPS

1.1. Видове инструкции

- инструкции за работа с паметта: load и store

Примери:

lw $16, 100($2) значение: Reg16 ← Mem[Reg2 + 100]
sw $17, 200($2) значение: Mem[Reg2 + 200] ← Reg17

- инструкции за аритметични и логически операции с цели числа

Примери:

addi $8, $16, 17 значение: Reg8 ← Reg16 + 17
slt $10, $8, $9 значение: if(Reg2 < Reg3) Reg1←1; else Reg1←0

- инструкции за аритметични операции върху числа с плаваща запетая

Примери:

add.d $f0, $f1, f0 значение: Regf0 ← Regf1 + Regf0

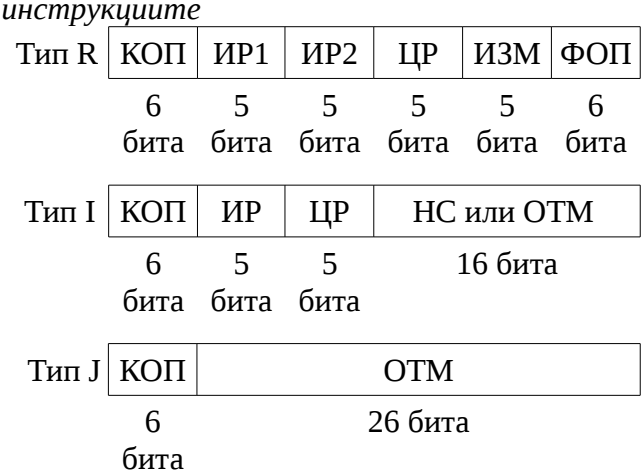
- инструкции за управление на последователността на изпълнение в програмата (условни и безусловни преходи)

Примери:

beq $0, $1, loop значение: if (Reg0 == Reg1) go to loop
j label1 значение: go to label1

- други инструкции

1.2. Формат на инструкциите



КОП - 6-битов код на операцията

ФОП - Функционален операционен код. Вариант на функцията, изпълнявана от инструкцията.

ЦР - Целеви регистър, в който ще се запише резултатът от операцията.

ИР - Изходен регистър, от който ще се вземе стойността на входен операнд.

ИЗМ - Изместване. Показва с колко бита да се измести стойността е ИР.

НС - Непосредствена стойност. Константа записана вътре в кода на самата инструкция.

ОТМ - Отместване. Добавя се към програмния брояч (PC), за да се формира адреса на прехода.

Тип R – Аритметични инструкции

КОП	ИР1	ИР2	ЦР	ИЗМ	ФОП
6 бита	5 бита	5 бита	5 бита	5 бита	6 бита

Тип I – Инструкции load/store

КОП	ИР	ЦР	НС или ОТМ
6 бита	5 бита	5 бита	16 бита

Тип J – Инструкции за преход

КОП	ОТМ
6 бита	26 бита

1.3. Опростен набор инструкции

■ Всички операции с данни се отнасят за данните в регистрите и обикновено променят целия регистър (32 бита в регистър).

■ Единствените операции, които засягат паметта, са операциите за зареждане и съхраняване, които преместват данни от паметта в регистър или съответно от регистър в паметта. Често са налични операции за зареждане и съхраняване, които зареждат или съхраняват непълен регистър (например байт или 16 бита).

■ Форматите на инструкциите са малко на брой, като всички инструкции обикновено са с един размер. В MIPS, например, регистровите спецификатори: ИР1, ИР2 и ЦР са винаги на едно и също място, опростявайки управлението.

Тези характеристики водят до значително опростяване на реализацията на конвейерното изпълнение, което и обяснява защо тези системи инструкции са проектирани по този начин.

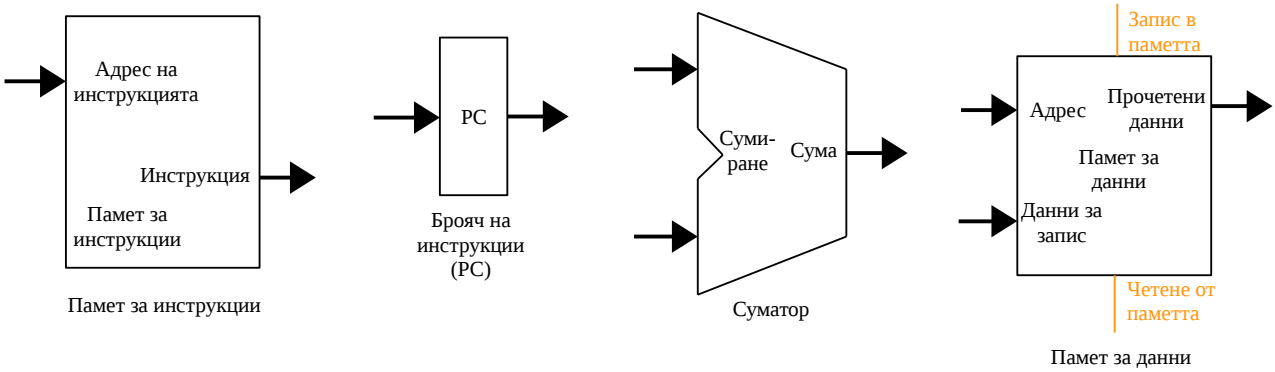
1.4. Регистри и тяхното стандартно предназначение

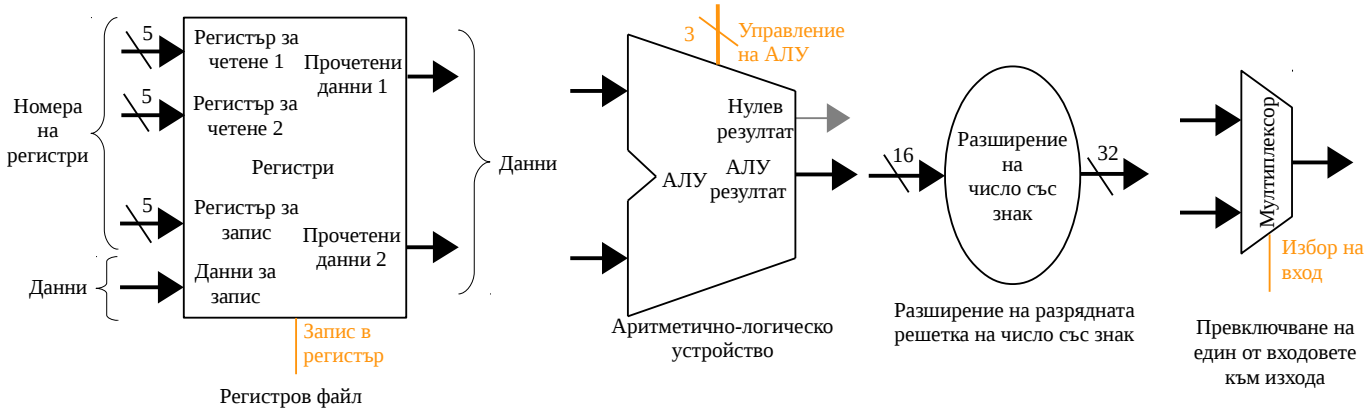
Регистрите в MIPS имат стандартно предназначение, но за повечето от тях то не е строго фиксирано.

Име	***Рег. №***	***Употреба***
zero	0	константа 0
at	1	запазен за програмата за асемблиране
v0-v1	2-3	върнати стойности от извикани процедури
a0-a3	4-7	аргументи предавани към извикани процедури
t0-t7	8-15	временни стойности
s0-s7	16-23	съхранени стойности
t8-t9	24-25	временни стойности
k0-k1	26-27	запазени за ядрото на операционната система
gp	28	указател за адресиране на глобална памет
sp	29	указател за адресиране на стек
fp	30	указател за адресиране на кадър
ra	31	адрес на връщане

2. Изпълнение на инструкциите

2.1. Цифрови компоненти, чрез които се реализира изпълнението на инструкциите





2.2. Степени или етапи при изпълнение на инструкциите

За да се разбере как една система инструкции може да се реализира в MIPS процесор, прилагайки конвейерно изпълнение, първо трябва да се разбере как процесора изпълнява инструкциите без да се използва конвейер. В тази точка е показана реализация, при която всяка инструкция отнема най-много 5 процесорни цикъла. Тази неконвейерна реализация не е нито най-икономичната, нито най-ефективната. Вместо това, тя е проектирана да доведе по естествен начин до конвейерна реализация. Реализацията на набора от инструкции изисква въвеждането на няколко временни регистри, които не са част от архитектурата; те не са представени в този раздел за опростяване на конвейерната реализация. Представената реализация е съсредоточена само върху изпълнение за подмножество от MIPS системата инструкции, отнасящо се до целочислени изчисления, която се състои от операции за работа с паметта (зареждане и съхранение), инструкции за преходи в програмата и аритметично-логически операции за работа с цели числа.

Всяка инструкция в това подмножество на MIPS архитектурата може да бъде изпълнена в, най-много, 5 процесорни цикъла, както следва:

1. Цикъл за **извличане на инструкция** (IF):

Програмният брояч (PC) сочи към адрес в паметта от където да се извлече текущата инструкция. Програмният брояч се актуализира като се добавя 4 (тъй като всяка инструкция е с дължина 4 байта), за да сочи към следващия адрес в паметта, където се намира поредната инструкция от програмата.

2. Цикъл за **декодиране на инструкция/извличане на стойностите (операндите) от регистри** (ID):

Декодира инструкцията и чете от регистрите в регистровия файл, съответстващи на регистрите източници, зададени в спецификаторите на инструкцията.

Декодирането се извършва паралелно с четенето от регистрите. Това е възможно, защото спецификаторите указващи регистрите са на фиксирано място в инструкциите. Тази техника е известна като „fixed-field decoding”. Трябва да се има предвид, че може да се случи а се прочете регистър, който не се използва, което не е необходимо, но и не вреди на работата. (В този случай се използва енергия за четене на ненужна стойност от регистър, което може да се избегне в проектите изискващи енерго-ефективност.) Непосредствените полета в някой от операциите, напр. зареждане (load) и аритметично-логическите операции (ALU) с непосредствени полета, винаги са на едно и също място, така че лесно може да се извърши преобразуване/разширение на стойностите със знак (от 16 към 32 битово представяне).

3. Цикъл **изпълнение/определяне на ефективен адрес** (EX):

ALU работи с операндите, подготвени в предходния цикъл, изпълнявайки една от трите функции, в зависимост от типа инструкция.

■ Адресиране на паметта - АЛУ добавя отместването към базовия регистър, за да образува ефективния адрес.

■ Инструкции регистър-регистър на ALU - АЛУ извършва операцията, посочена от кода на операцията (КОП) върху стойностите, прочетени от регистровия файл.

■ Инструкции регистър-литерал (непосредствена стойност) на ALU - АЛУ извършва операцията, посочена от кода на операцията (КОП) на първата стойност, прочетена от регистровия файл и непосредствената, разширена със знак стойност (литералът).

■ Условен преход - Определя дали условието е вярно. Прави проверка за равенство на регистрите, когато се изпълнява инструкция за преход. В случай че е необходимо, използва полето за отместване в инструкцията, което съдържа и знак (+/-). Изчислява възможния целеви адрес на прехода, като инкрементира програмния брояч с отместването.

В архитектурата “load-store” циклите „определяне на ефективния адрес” и „изпълнение„ могат да бъдат обединени и изпълнени в един тактов цикъл, тъй като не е необходимо едновременно да се изчислява адреса на данните и да се изпълнява операцията върху тях.

4. Достъп до паметта (MEM):

Ако инструкцията е за зареждане (load) се извършва четене от паметта, като се използва ефективния адрес, изчислен в предходния цикъл. Ако инструкцията е за съхраняване (store), тогава данните, прочетени от втория регистър от регистровия файл, се записват в паметта, използвайки ефективния адрес.

5. Цикъл на обратно записване (WB):

■ Инструкции регистър-регистър на ALU или инструкция за зареждане (load):

Записва резултата в регистровия файл, независимо дали е прочетен от паметта (за инструкцията зареждане - load) или е получен от АЛУ (за инструкция на ALU).

И така накратко степените (етапите) в изпълнение на инструкциите са:

I. *IF* – Извличане на инструкцията от паметта за инструкции според адреса подаден от програмният брояч (PC); PC нараства с 4, за да сочи следващата инструкция в паметта.
II. *ID* – Декодиране на инструкцията; Четене на операндите от регистрите в регистровия файл.

- lw $a, 100($b)

III. *EX* – Събиране на стойността в регистър $b със 100
IV. *MEM* – Четене от паметта
V. *WB* – Запис в регистър $a

- sw $a, 100($b)

III. *EX* – Събиране на стойността в регистър $b със 100
IV. Запис в паметта

- add $c, $a, $b

III. *EX* – Събиране на стойностите в регистри $a и $b
IV. *MEM* – Не се извършва нищо. Изчаква се да завърши процесорния цикъл
V. *WB* – Запис в регистър $c

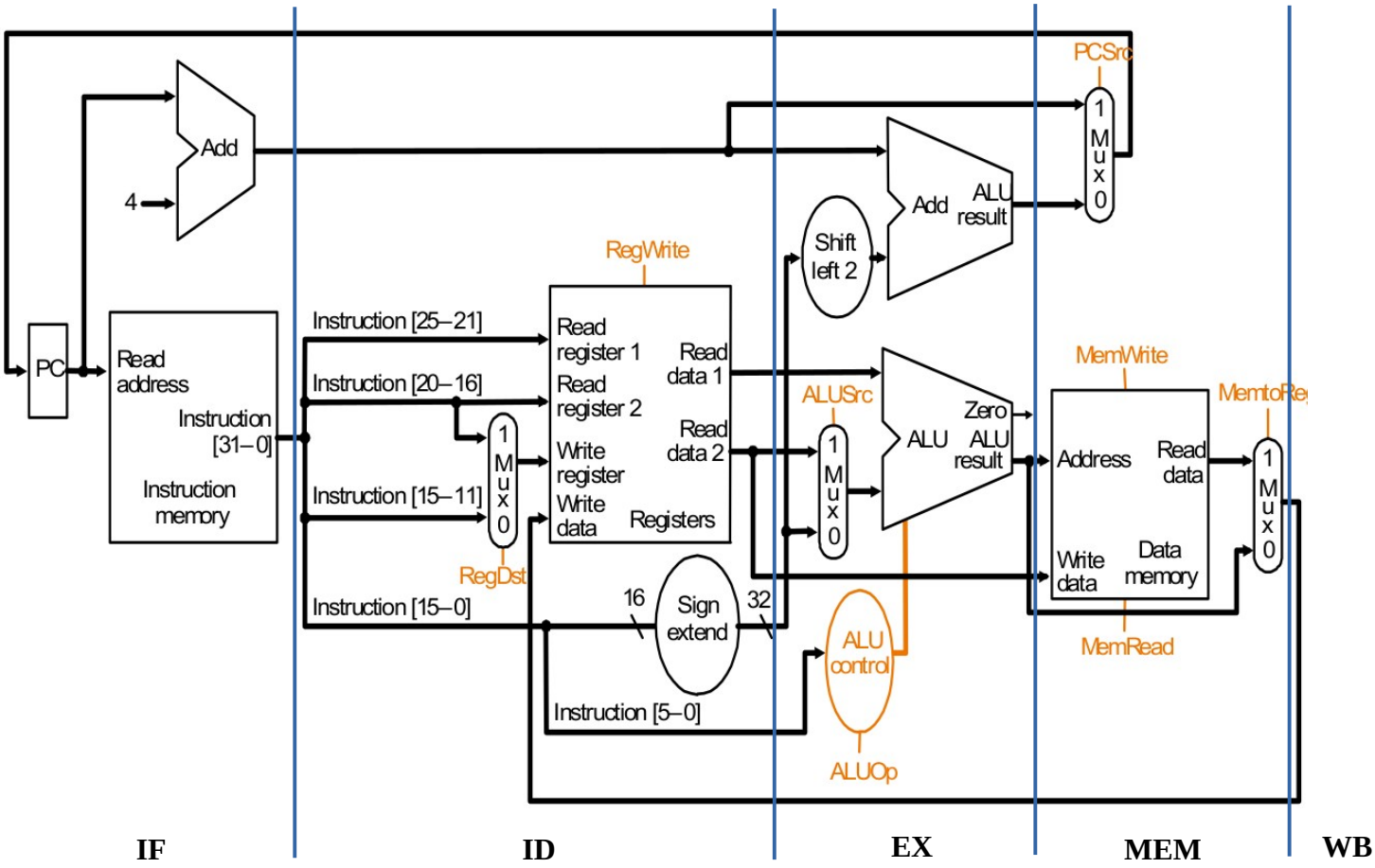
- beq $a, $b, offset

III. *EX* – Сравнява стойностите в регистри $a и $b. Ако са равни, PC ← PC + offset

Не всички инструкции се изпълняват за 5 процесорни цикъла.

Инструкция load	**Инструкция store**	**Инструкции за аритметични и логически операции (ALU)**	**Инструкции за преход (branch)**
IF	IF	IF	IF
ID	ID	ID	ID
EX	EX	EX	EX
MEM	MEM	--	--
WB	--	WB	--

2.3. Блокова схема на устройството за изпълнение на инструкции в процесора



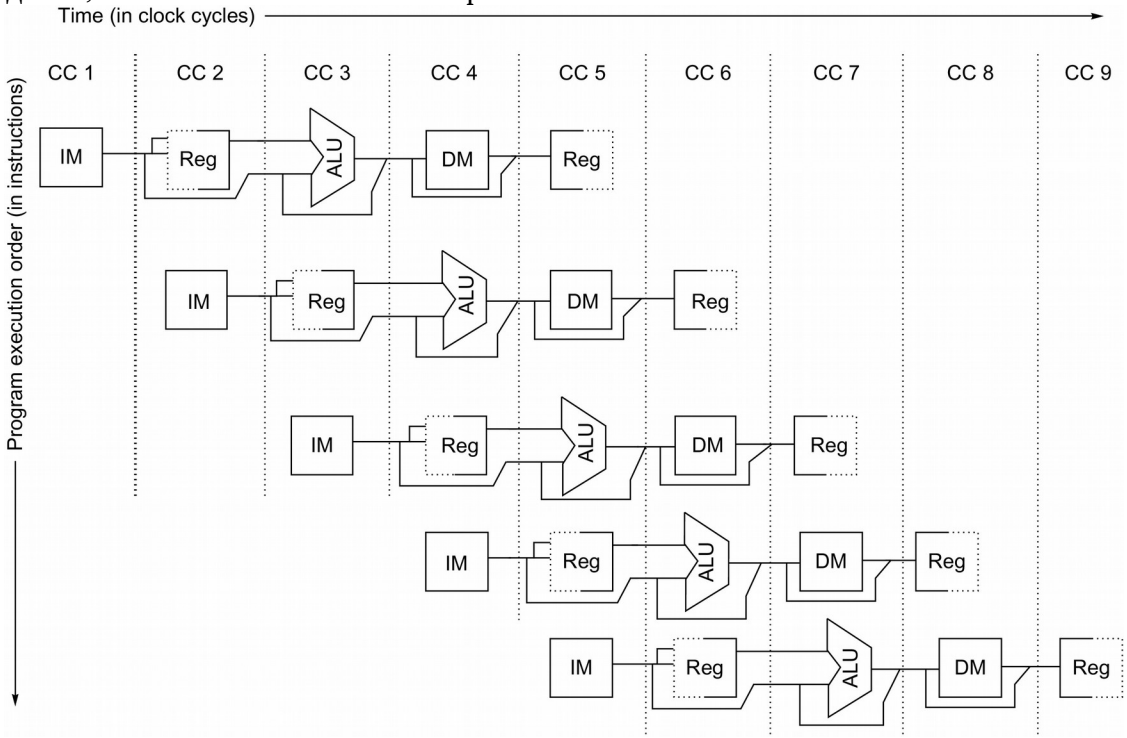
3. Конвейерно изпълнение на инструкциите

Вместо да се изчаква докато една инструкция да завърши изпълнението си преминавайки през петте процесорни цикъла и тогава да се стартира изпълнението на следващата може да се стартира нова инструкция веднага след като предходната завърши първия процесорен цикъл. Така на всеки процесорен цикъл ще се стартира нова инструкция. Всеки един от процесорните цикли описани в предишния раздел става етап или цикъл в конвейера. Това води до схемата на изпълнение, показана на следващата фигура. Въпреки че всяка инструкция отнема 5 процесорни цикъла, с всеки нов процесорен цикъл хардуерът ще започне изпълнението на нова инструкция и ще изпълнява по една част от пет различни инструкции.

Инструкция №	**Процесорен цикъл**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Инструкция i	IF	ID	EX	MEM	WB				
Инструкция i+1		IF	ID	EX	MEM	WB			
Инструкция i+2			IF	ID	EX	MEM	WB		
Инструкция i+3				IF	ID	EX	MEM	WB	
Инструкция i+4					IF	ID	EX	MEM	WB

На всеки процесорен цикъл нова инструкция се извлича и започва своето изпълнение. Тъй като на всеки процесорен цикъл една инструкция започва своето изпълнение, то и производителността на процесора ще бъде пет пъти по-голяма от тази на процесор без конвейерно изпълнение. Имената на етапите в конвейера са същите като на отделните цикли в неконвейерната реализация: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, и WB = write-back.

Представено така конвейерното изпълнение на инструкциите изглежда много просто. Има обаче някой особености, които трябва да бъдат отчетени при реализацията на реален процесорен конвейер. Първоначално трябва да се определи какво се случва на всеки процесорен цикъл, за да се провери дали в един и същ процесорен цикъл не се извършват две операции с един и същ изчислителен ресурс. Например, едно АЛУ не може едновременно да извърши изчисление на ефективен адрес и да извърши операция изваждане.
На следващата фигура е показана опростена версия на изчислителните ресурси в MIPS архитектура, реализираща конвейерно изпълнение. Основните функционални единици (изчислителни ресурси) се използват в различни цикли и следователно припокриването на изпълнението на множество инструкции продуцира сравнително малко конфликти. Има три наблюдения, на които се основава този факт.



Фигурата показва застъпването между функционалните елементи, реализиращи изпълнението на инструкциите. В процесорен цикъл 5 (CC 5) е показана ситуация в която регистровия файл се появява два пъти – веднъж се използва като източник на данни в етапа ID и втори път се използва за запис на резултата от инструкцията в етапа WB. Четенето е представено с прекъсната линия в лявата част и непрекъсната в дясната, а записа е представен с непрекъсната линия в лявата част и прекъсната в дяната. Използваните абревиатури са: IM = instruction memory, DM = data memory, Reg = registries (register file), ALU = arithmetic-logic unit и CC = clock cycle.

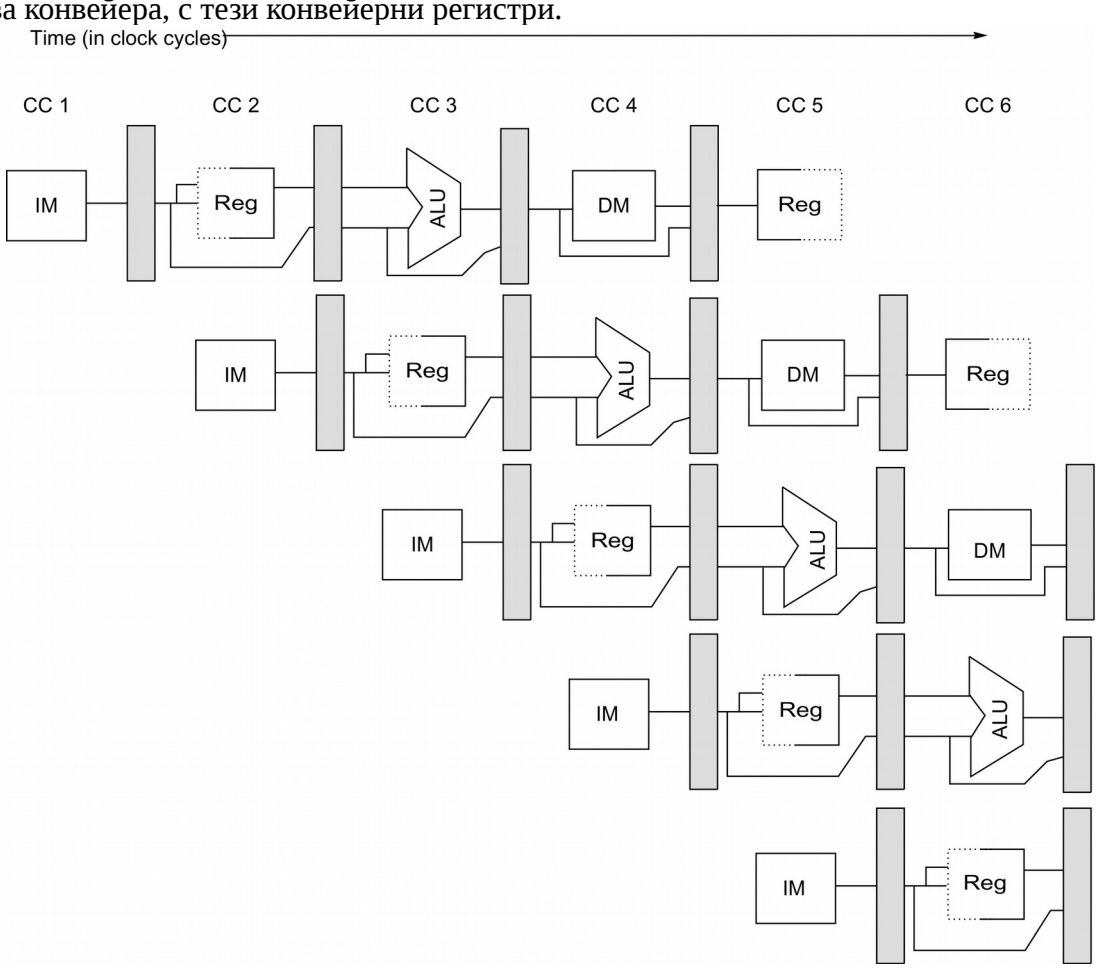
Първо, използваме отделни памети за инструкции и данни, които обикновено се реализират с отделни кеш-памети за инструкции и данни. Използването на отделни кеш памети елиминира конфликта, който би възникнал между извличането на инструкция и достъпа до паметта за данни, ако паметта беше обща. Забележете, че ако нашият конвейерен процесор има тактова честота равна на тази на неконвейерната версия, паметта трябва да

осигури пет пъти по-голяма пропускателна способност. Това изискване е цената за по-високата производителност.

Второ, регистровият файл се използва в два от етапите: веднъж за четене в етапа ID и втори път за запис в етапа WB. Това са два различни начина за използване на регистровия файл, което позволява неговото използване и на двете места в конвейера. Необходимо е да се извършат две прочитания и един запис при всеки тактов цикъл. За обработката на четенето и записа в един и същ регистър (и по друга причина, която ще стане очевидна скоро), записването се извършва през първата половина на тактовия цикъл, а четенето през втората.

Трето, на горната фигура не е представен програмния брояч PC. За стартирането на нова инструкция на всеки цикъл, трябва да увеличим и съхраним PC също на всеки процесорен цикъл и това трябва да стане по време на етапа IF при подготовката за следващата инструкция. Освен това трябва да имаме и суматор, който да изчисли потенциалния целеви адрес ако възникне преход по време на етапа ID. Друг проблем е, че е необходимо ALU в етапа ALU, за да се изчисли условието за преход. Всъщност не е необходимо пълноценно ALU, за да се сравнят два регистъра, но е необходима достатъчно функционалност, за извършване на операциите, които могат да възникнат на този етап в конвейера.

Важно е да се гарантира, че инструкциите в конвейера не се опитват да използват хардуерните ресурси едновременно. Също така, трябва да се гарантира, че инструкциите в различни етапи на конвейера не си влияят една на друга. Това разделяне се извършва чрез въвеждане на конвейерни регистри между съседните етапи в конвейера, така че в края на тактовия цикъл всички резултати от даден етап да се съхраняват в регистър, който се използва като вход към следващия етап при следващия тактов цикъл. Следващата фигура показва конвейера, с тези конвейерни регистри.



Въпреки че в много фигури тези регистри се пропускат за простота, те са необходими, за да работи правилно конвейера. Разбира се, подобни регистри са необходими, дори в

многоциклично изчислително устройство без конвейерна обработка, за да се съхраняват междинните стойности при преходите към следващия тактов период. При конвейерните процесори, конвейерните регистри също играят ключова роля за пренасяне на междинните резултати от един етап към друг, където източникът и местоназначението може да не са съседни. Например, за да се съхрани стойността на регистър, по време на инструкция за съхранение (store), тя се прочита по време на етапа ID, но всъщност не се използва, докато не се достигне етапа MEM; тя се предава през два конвейерни регистъра, за да бъде записана в паметта за данните по време на етапа MEM. По същия начин резултатът от една ALU инструкция се изчислява по време на етапа EX, но действително се съхранява при достигане на етапа WB, минавайки през два конвейерни регистъра. Понякога е полезно да се посочат с имена конвейерните регистри. Имената им се образуват от етапите, между които се намират регистрите: IF/ID, ID/EX, EX/MEM и MEM/WB.

4. Състезанията между инструкциите в конвейера

Представения MIPS конвейер ще функционира отлично за целочислени инструкции, ако всяка инструкция бъде независима от всяка друга инструкция в конвейера. В действителност инструкциите в един конвейер могат да зависят една от друга; това е темата на следващия раздел.

Основното препятствие пред реализацията на конвейерната обработка – състезанията между инструкциите в конвейера

Има ситуации, наречени състезания, които препятстват изпълнението на следващата инструкция в последователността от инструкции по време на определения ѝ тактов период. Състезанията намаляват производителността спрямо максималното бързодействие, което може да се получи от използването на конвейерна обработка. Има три типа състезания:

1. **Структурни състезания** – възникват при наличие на конфликти за заемане на ресурси, в случай, че хардуерът не може да поддържа всички възможни комбинации от инструкции едновременно при припокриването на тяхното изпълнение. В съвременните процесори структурните състезания възникват предимно във функционалните елементи със специално предназначение, които се използват по-рядко (като деление с плаваща запетая или други сложни инструкции с продължително изпълнение). Те не са определящ фактор за производителността, вземайки предвид, че програмистите и авторите на компилатори са запознати с по-ниската производителност на тези инструкции. Този вид състезания възникват по-рядко, затова се съсредоточаваме върху другите два типа състезания, които възникват много по-често.

2. **Състезания за данни** – възникват, когато една инструкция зависи от резултата от предишна инструкция поради припокриването на изпълнението им в конвейера.

3. **Състезания за управление** (на последователността на изпълнението) – възникват от конвейеризирането на инструкциите за преход и други инструкции, които променят програмния брояч (PC).

Предотвратяването на състезанията в конвейера налага да се използва, т.нар. „застой” – изпълнението на някои инструкции в конвейера продължава, докато на други се забавя. Когато дадена инструкция е в застой, всички следващи инструкции също са в застой. Инструкциите, предшестващи спряната трябва да продължат. В противен случай състезанието никога не би се прекратило. По време на „застоя” не се извличат нови инструкции.

В следващата лекция ще видите различни примери „застои“ при състезания между инструкциите в конвейера и начините за тяхното намаляване. Не се притеснявайте, не е толкова сложно, колкото звучи.